Types

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

Preservation If $\Gamma \vdash e : \tau$ and $\exists e'.\ e \rightsquigarrow e'$ then $\Gamma \vdash e' : \tau$.

PLC

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

Existentials

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

$$\frac{\Theta; \Gamma \vdash e_1 : \exists\alpha.\ A \quad \Theta, \alpha; \Gamma, x : A \vdash e_2 : B}{\Theta; \Gamma \vdash \text{let pack}(\alpha, x) = e_1 \text{ in } e_2 : B}$$

Monads

Relation $\div$ for effectful statements, can be encapsulated and passed around pure code using $\{e\}$ which places effectful expressions in a monad T.

$$\frac{\Sigma; \Gamma \vdash e : A}{\Sigma; \Gamma \vdash \text{return } e \div A} \qquad \frac{\Sigma; \Gamma \vdash e_1 : TA \quad \Sigma; \Gamma, x : A \vdash e_2 \div B}{\Sigma; \Gamma \vdash \text{let } x = e_1; e_2 \div B}$$

$$\frac{\Sigma; \Gamma \vdash e \div A}{\Sigma; \Gamma \vdash \{e\} : TA}$$

Classical Logic

To prove something holds, use a $\mu x : A.\ c$ term to **assume** A **with proof** x **and derive a contradiction** c. Contradiction terms $\langle e \mid_B k \rangle$ show B is true with e and false with k.

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

$$\frac{\Gamma; \Delta \vdash e : B \text{ true} \quad \Gamma; \Delta \vdash k : B \text{ false}}{\Gamma; \Delta \vdash \langle e \mid_B k \rangle \text{ contr}}$$

Data

PLC can represent datatypes as functions – no need for axiomatic definitions. Eg.

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

1 Proof

Structural Induction

Proof based on the rules, so 'structural induction on $\vdash$' not 'on $\Gamma \vdash e : \tau$'.

For each rule, the 'case' is the rule. We assume that both the judgements on the top hold (as they're subterms) and that the hypothesis holds **on those terms**.

2 Terminology

Congruence Rule
A rule for $\rightsquigarrow$ that reduces a subterm, like $e_1\ e_2 \rightsquigarrow e_1'\ e_2$.

Reduction Rule
A rule for $\rightsquigarrow$ that transform the term, like $(\lambda x : X.\ e)\ v \rightsquigarrow e[v/x]$.

3 Theorems

Weakening

If $\Gamma, \Gamma' \vdash e : \tau$ then $\Gamma, x : \tau'', \Gamma' \vdash e : \tau$.

If a term typechecks under a context, then it typechecks under a larger context.

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 typechecks in a reordered context.

Substitution

If $\Gamma \vdash e : \tau$ and $\Gamma, x : \tau \vdash e' : \tau'$ then $\Gamma \vdash [e/x]e' : \tau'$.

Substituting a type-correct term for a variable preserves type-correctness.

Progress

If $\cdot \vdash e : \tau$ then either e is a value or $\exists e'.\ e \rightsquigarrow e'$.

Type-correct terms are never stuck. Note that it's $\cdot$ not any Γ.

Preservation

If $\cdot \vdash e : \tau$ and $\exists e'.\ e \rightsquigarrow e'$ then $\cdot \vdash e' : \tau$.

Evaluation preserves types (evaluation may be non-terminating). Note that it's $\cdot$ not any Γ.

Proof by structural induction on $e \rightsquigarrow e'$.

Determinacy

If $e \rightsquigarrow e'$ and $e \rightsquigarrow e''$ then $e' = e''$.

Proof by structural induction on $e \rightsquigarrow e'$.

Type Safety Progresss + Preservation

Consistency

There are no proofs of $\bot$, or there are no terms e such that $\cdot \vdash e : 0$.

Possible in all good logics, but only possible in **total** languages.

Curry-Howard Correspondence

Programs are proofs, types are formulae. Not an isomorphism, consider $X \implies (Y \implies X \wedge Y)$.

Type Weakening If $\Theta, \Theta' \vdash A$ type then $\Theta, \beta, \Theta' \vdash A$ type.

Type Exchange If $\Theta, \beta, \gamma, \Theta' \vdash A$ type then $\Theta, \gamma, \beta, \Theta' \vdash A$ type.

Type Substitution If $\Theta \vdash A$ type and $\Theta, \alpha \vdash B$ type then $\Theta \vdash B[A/\alpha]$ type.

Context Weakening If $\Theta, \Theta' \vdash \Gamma$ ctx then $\Theta, \alpha, \Theta' \vdash \Gamma$ ctx.

Context Exchange If $\Theta, \beta, \gamma, \Theta' \vdash \Gamma$ ctx then $\Theta, \gamma, \beta, \Theta' \vdash \Gamma$ ctx.

Context Substitution If $\Theta \vdash A$ type and $\Theta, \alpha \vdash \Gamma$ ctx then $\Theta \vdash \Gamma[A/\alpha]$ ctx.

Regularity

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

Typechecking only succeeds for well-formed types.

Proof by structural induction on $\Theta; \Gamma \vdash e : A$.

4 Recursive Functions

$$v ::= ... \mid \text{fun}_{X \to Y}\ f\ x.\ e$$

$$\frac{\Gamma, f : X \to Y, x : X \vdash e : Y}{\Gamma \vdash \text{fun}_{X \to Y}\ f\ x.\ e : X \to Y}\,(\text{Fix}) \qquad \frac{}{(\text{fun}_{X \to Y}\ f\ x.\ e)v \rightsquigarrow e[v/x, (\text{fun}_{X \to Y}\ f\ x.\ e)/f]}$$

Natural for defining functions: $\text{fun}_{\texttt{int} \to \texttt{int}}\ fact\ n.$ if $n = 0$ then 1 else $n \times fact(n-1)$. Also allows for infinite recursion, so no longer total/consistent.

5 The Halt Relation

e halts iff $\exists v.\ e \rightsquigarrow^* v$.

Define a type-indexed family of sets of terms:

- $\text{Halt}_0 = \emptyset$: there are no terms of type 0 that halt.
- $e \in \text{Halt}_1$ only when e halts and $\cdot \vdash e : 1$.
- $e \in \text{Halt}_{X \to Y}$ only when e halts, $\cdot \vdash e : X \to Y$, and $\forall e'.\ e' \in \text{Halt}_X \implies (e\ e') \in \text{Halt}_Y$.

All terms in Halt_1 halt, all terms in $\text{Halt}_{X \to Y}$ are functions which preserve the property of halting, ...

Halting

If $e \in \text{Halt}_X$ then e halts ($\exists v.\ e \rightsquigarrow^* v$).

Closure Lemma

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

Proof by induction on the 'type indices' to the sets, ie. by structural induction where we need to prove the lemma for the types $X = 1, Y \to Z, ...$, assuming for the IH that Halt_t holds for all subtypes t of X.

The Fundamental Lemma

If $\forall n \geq 0$, $x_1 : X_1, ..., x_n : X_n \vdash e : Z$ and $\forall i \in [1..n]$, $\cdot \vdash v_i : X_i$ and $v_i \in \text{Halt}_{X_i}$, then $[v_1/x_1, ..., v_n/x_n]e \in \text{Halt}_Z$.

Proof by structural induction on $x_1 : X_1, ..., x_n : X_n \vdash e : Z$ (go through the rules).

This is powerful as it says that *all well-typed terms terminate.*

Consistency

There are no terms e such that $\Gamma \vdash e : 0$. Proof by contradiction using the fundamental lemma:

Assume $\Gamma \vdash e : 0$, then by the fundamental lemma (with $n = 0$) $e \in \text{Halt}_0$. But by the definition of Halt_0, $e \notin \text{Halt}_0$, so contradiction.

6 Halt for System F

Similar to Halt for STLC, but uses a semantic interpretation to define the sets. Probably too complex to be expected to remember, rules would be given in exam.

7 Datatypes

7.1 STLC

Builtin	Encoding
Bool	$1 + 1$
True	L $\langle\rangle$
False	R $\langle\rangle$
if e then e' else e''	case$(e, \text{L } _ \to e', \text{R } _ \to e'')$
Char	Bool^7
'A'	(True,False,False,False,False,False,True)
...	...

7.2 PLC

Builtin	Encoding
Bool	$\forall\alpha.\ \alpha \to \alpha \to \alpha$
True	$\Lambda\alpha.\ \lambda x : \alpha.\ \lambda y : \alpha.\ x$
False	$\Lambda\alpha.\ \lambda x : \alpha.\ \lambda y : \alpha.\ y$
if e then e' else $e'' : X$	$e\ X\ e'\ e''$
$X \times Y$	$\forall\alpha.\ (X \to Y \to \alpha) \to \alpha$
$\langle e, e' \rangle$	$\Lambda\alpha.\lambda k : X \to Y \to \alpha.\ k\ e\ e'$
fst e	$e\ X\ (\lambda x : X.\ \lambda y : Y.\ x)$
snd e	$e\ X\ (\lambda x : X.\ \lambda y : Y.\ y)$
$X + Y$	$\forall\alpha.\ (X \to \alpha) \to (Y \to \alpha) \to \alpha$
L e	$\Lambda\alpha.\ \lambda f : (X \to \alpha).\ \lambda g : (Y \to \alpha).\ f\ e$
R e	$\Lambda\alpha.\ \lambda f : (X \to \alpha).\ \lambda g : (Y \to \alpha).\ g\ e$
case$(e, \mathrm{L}\ x \to e_1, \mathrm{R}\ y \to e_2) : Z$	$e\ Z\ (\lambda x : (X \to Z).\ e_1)\ (\lambda y : (Y \to Z).\ e_2)$
$\mathbb{N}$	$\forall\alpha.\ \alpha \to (\alpha \to \alpha) \to \alpha$
z	$\Lambda\alpha.\ \lambda z : \alpha.\ \lambda s : (\alpha \to \alpha).\ z$
s(e)	$\Lambda\alpha.\ \lambda z : \alpha.\ \lambda s : (\alpha \to \alpha).\ s\ (e\ \alpha\ z\ s)$
iter$(e, \mathrm{z} \to e_z, \mathrm{s}(x) \to e_s) : X$	$e\ X\ e_z\ (\lambda x : X.\ e_s)$
...	...

The function(s) 'attached' to the encoded datatypes are selection operations: for Bool it's switching, for pairs it's selecting, for sums it's conditional.

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

8 Existentials

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

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

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

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

α is a placeholder for the concrete type, C is the *concrete* type, and A is the *abstract* interface.

- $\text{pack}_{\alpha,A}(C, e)$ packs a implementation e with concrete type C into a provider for the α, A interface.
- let $\text{pack}(\alpha, x) = \text{pack}_{\alpha,A}(C, v)$ in e unpacks a provider for the α, A interface into the concrete type and implementation α and x.

Encoding

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

9 State

$$
\begin{array}{lll}
\text{Types} & X & ::= \ldots \mid \text{ref } X \\
\text{Terms} & e & ::= \ldots \mid \text{new } e \mid\ !e \mid e := e' \mid l \\
\text{Values} & v & ::= \ldots \mid l \\
\text{Stores} & \sigma & ::= \cdot \mid \sigma, l : v \\
\text{Store Typings} & \Sigma & ::= \cdot \mid \Sigma, l : X
\end{array}
$$

$$
\frac{\Sigma; \Gamma \vdash e : X}{\Sigma; \Gamma \vdash \text{new } e : \text{ref } X} \qquad \frac{l : X \in \Sigma}{\Sigma; \Gamma \vdash l : \text{ref } X}
$$

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

Operational semantics are for $\langle \sigma, e \rangle$.

There's a store well-typing rule but the notes mess it up: essentially for each store element, compare the type inferred from the value to the type stored in the store context Σ.

Preservation

The normal preservation property no longer holds:

$e = \langle \cdot; \text{new } \langle\rangle \rangle : \langle \cdot; \text{ref } 1 \rangle$, but $e \rightsquigarrow \langle l : \langle\rangle; l \rangle : \langle l : 1; \text{ref } 1 \rangle$, where the stores don't match.

Store Extension

Define $\Sigma \leq \Sigma'$ to mean $\exists \Sigma''.\ \Sigma' = \Sigma, \Sigma''$.

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 $\Sigma' \vdash \sigma_0 : \Sigma_0$.

Allocating new references never breaks term typability.

Preservation Repaired

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

Recursion

With state we can perform recursion even in the STLC, by storing and updating a function in a reference.

10 Monads

Separate judgements between pure and impure computations, to prevent pure code being able to perform impure.

$$\begin{array}{lll} \text{Types} & X & ::= \ldots \mid \text{ref } X \mid T\ X \\ \text{Pure Terms} & e & ::= \ldots \mid l \mid \{t\} \\ \text{Impure Terms} & t & ::= \text{new } e \mid\ !e \mid e := e' \mid \text{let } x = e;\ t \mid \text{return } e \\ \text{Values} & t & ::= \ldots \mid l \mid \{t\} \end{array}$$

$$\frac{\Sigma; \Gamma \vdash t \div X}{\Sigma; \Gamma \vdash \{t\} : T\ X}$$

$$\frac{\Sigma; \Gamma \vdash e : \text{ref } X}{\Sigma; \Gamma \vdash\ !e \div X} \qquad \frac{\Sigma; \Gamma \vdash e : X}{\Sigma; \Gamma \vdash \text{return } e \div X} \qquad \frac{\Sigma; \Gamma \vdash e : T\ X \quad \Sigma; \Gamma, x : X \vdash t \div Z}{\Sigma; \Gamma \vdash \text{let } x = e;\ t \div Z} \qquad \ldots$$

let $x = e;\ t \div Z$ evaluates a monadic effect within other impure code. **No escape from the monad**, though.

All the impure terms are typed by $\div$: no introduction of the T monad yet. The $\{t\}$ term in the pure terms converts a judgement of $\div$ into a monadic type so it can be passed around pure code without tainting it.

Proofs are by mutual induction, as there are distinct properties for pure and impure terms.

11 Classical Logic

Logic

Step away from the intuitionistic view of refutations being deriving ($\neg P \stackrel{\text{def}}{=} P \Longrightarrow \bot$) by making them first-class.

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

A proof is $\Gamma; \Delta \vdash A$ true: if things in Γ are true and things in Δ are false, then A is true. Similarly for $\Gamma; \Delta \vdash A$ false.

A refutation is $\Gamma; \Delta \vdash \text{contr}$, when Γ and Δ contradict one another.

Note that there's no implication: can derive it as $A \Longrightarrow B \stackrel{\text{def}}{=} \neg A \vee B$.

Proofs

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

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

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

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

Refutations

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

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

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

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

These allow proving **most** of classical logic, but not all: we can't prove $\neg\neg A; \cdot \vdash A$, or $A \wedge B \Longrightarrow A$.

Introduce contradiction rules:

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

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

Language

STLC + continuations. Essentially mirrors the logic:

$$\begin{array}{lll}
\text{Propositions} & X & ::= \top \mid \bot \mid X \wedge Y \mid X \vee Y \\
\text{True Contexts} & \Gamma & ::= \cdot \mid \Gamma, x : A \\
\text{False Contexts} & \Delta & ::= \cdot \mid \Delta, u : A
\end{array}$$

$$\begin{array}{lll}
\text{Expressions} & e & ::= \langle\rangle \mid \langle e, e' \rangle \mid \text{L } e \mid \text{R } e \mid \text{not}(k) \mid \mu u : A.\ c \\
\text{Continuations} & k & ::= [] \mid [e, e'] \mid \text{fst } e \mid \text{snd } e \mid \text{not}(e) \mid \mu x : A.\ c \\
\text{Contradictions} & c & ::= \langle e \mid_A k \rangle
\end{array}$$

Proof Terms

$$\frac{x : A \in \Gamma}{\Gamma; \Delta \vdash x : A \text{ true}} \qquad \frac{}{\Gamma; \Delta \vdash \langle\rangle : \top \text{ true}}$$

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

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

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

Refutation Terms

$$\frac{x : A \in \Delta}{\Gamma; \Delta \vdash x : A \text{ false}} \qquad \frac{}{\Gamma; \Delta \vdash [] : \bot \text{ false}}$$

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

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

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

Contradictions are also similar. Note that $\Gamma; \Delta \vdash c \text{ contr}$ is a completely different judgement than $\Gamma; \Delta \vdash c \text{ true}$.

A contradiction $\langle e \mid_A k \rangle$ is a term containing two terms which prove the truth and falsehood of a type A.

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

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

Operational semantics for contradictions $\langle e \mid_A k \rangle$:

$$\langle \langle e_1, e_2 \rangle \mid_{A \wedge B} \text{fst } k \rangle \rightsquigarrow \langle e_1 \mid_A k \rangle \qquad \langle \text{R } e \mid_{A \wedge B} [k_1, k_2] \rangle \rightsquigarrow \langle e \mid_A k_2 \rangle$$

$$\langle \langle e_1, e_2 \rangle \mid_{A \wedge B} \text{snd } k \rangle \rightsquigarrow \langle e_2 \mid_A k \rangle \qquad \langle \text{L } e \mid_{A \wedge B} [k_1, k_2] \rangle \rightsquigarrow \langle e \mid_A k_1 \rangle$$

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

$$\langle \mu u : A.\ c \mid_A k \rangle \rightsquigarrow c[k/u] \qquad \langle e \mid_A \mu x : A.\ c \rangle \rightsquigarrow c[e/x]$$

The term used in contradictions is normally something relatively complex that might already exist as an assumption. Before starting the contradiction, **break down the RHS** as much as possible: the smaller the thing you end up assuming for contradiction is, the more generally useful it is. For example:

$$\dfrac{\dfrac{\dfrac{\textit{...hard...}}{\neg(\neg A \wedge \neg B); A \vee B \vdash \text{contr}}}{\neg(\neg A \wedge \neg B); \cdot \vdash A \vee B \text{ true}}} \qquad \dfrac{\dfrac{\dfrac{\textit{...way easier...}}{\neg(\neg A \wedge \neg B); A \vee B \vdash \text{contr}}}{\neg(\neg A \wedge \neg B); \cdot \vdash A \text{ true}}}{\neg(\neg A \wedge \neg B); \cdot \vdash A \vee B \text{ true}}$$

Don't forget to use the **normal rules** like $\vee$ and $\wedge$.

To construct terms, normally easiest to prove the theorem using the logic, then mirror the tree in the syntax. For example:

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

A term proving this is then $\mu u : A.\ \langle x \mid_{A \wedge B} \mathrm{L}\ u \rangle$, where x is the assumed proof of $A \wedge B$. The typing derivation is:

$$\dfrac{\dfrac{}{x : A \wedge B; u : A \vdash x : A \wedge B \text{ true}} \quad \dfrac{\dfrac{}{x : A \wedge B; u : A \vdash u : A \text{ false}}}{x : A \wedge B; u : A \vdash \mathrm{L}\ u : A \wedge B \text{ false}}}{\dfrac{x : A \wedge B; u : A \vdash \langle x \mid_{A \wedge B} \mathrm{L}\ u \rangle \text{ contr}}{x : A \wedge B; \cdot \vdash (\mu u : A.\ \langle x \mid_{A \wedge B} \mathrm{L}\ u \rangle) : A \text{ true}}}$$

12 Embedding Classical Logic into Intuitionistic Logic

Translating from the continuation language into STLC.

Pick an arbitrary proposition p to act as false. Define $\sim X \stackrel{\text{def}}{=} X \to P$.

Can **translate** types from classical to intuitionistic using:

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

Note that $\sim\sim\sim X$ isn't $X \to X \to X \to p$, it's $((X \to p) \to p) \to p$.

The type $\neg\neg X \to X$ can't be translated into a constructive logic, but $\sim\sim\sim X \to \sim X$ can be (ignoring types for clarity):

$$\mathrm{tne} = \lambda f.\ \lambda a.\ f(\lambda g.\ g\ a)$$

We can construct a function $\mathrm{dne}_A : \sim\sim A^\circ \to A^\circ$ for every type A by using tne and structural induction (as we're talking about propositional logic, we have to define it for all the infinite types rather than quantifying):

$$\begin{aligned}
\mathrm{dne}_\top &= \lambda q : 1.\ \langle\rangle \\
\mathrm{dne}_\bot &= \lambda q : p.\ q\ (\lambda x.\ x) \\
\mathrm{dne}_{A \wedge B} &= \lambda q : (A^\circ \times B^\circ).\ \langle \mathrm{dne}_A(\mathrm{fst}\ p), \mathrm{dne}_B(\mathrm{snd}\ q) \rangle \\
\mathrm{dne}_{A \vee B} &= \lambda q : (\sim\sim \underbrace{\sim\sim(A^\circ \vee B^\circ)}_{(A \vee B)^\circ}).\ \mathrm{tne}\ q \\
\mathrm{dne}_{\neg A} &= \lambda q : \sim\sim \underbrace{\sim A^\circ}_{(\neg A)^\circ}.\ \mathrm{tne}\ q
\end{aligned}$$

13 Dependent Types

Types can include terms: merge the grammars, use judgements to decide if something is a type or a term.

Some terms constitute proofs: refl e is a proof that *something*?

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

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

Type Formation

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

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

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

Rules

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

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

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

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

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

Also rules defining $\equiv$ with normal equivalence relation rules, and congruence rules defining equality between terms. Gives power to the final rule given above, as it allows computation *within* types.

14 Languages

Simply Typed Lambda Calculus

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

$$\frac{}{\Gamma \vdash \langle\rangle : 1}\,(1_{\mathrm{I}}) \qquad \frac{\Gamma \vdash e : 0}{\Gamma \vdash \text{abort } e : X}\,(0_{\mathrm{E}}) \qquad \frac{x : X \in \Gamma}{\Gamma \vdash x : X}\,(\text{Hyp})$$

$$\frac{\Gamma, x : X \vdash e : Y}{\Gamma \vdash (\lambda x : X.\ e) : X \to Y}\,(\to_I) \qquad \frac{\Gamma \vdash e_1 : X \to Y \quad \Gamma \vdash e_2 : X}{\Gamma \vdash e_1\ e_2 : Y}\,(\to_E)$$

$$\frac{\Gamma \vdash e_1 : X \quad \Gamma \vdash e_2 : Y}{\Gamma \vdash \langle e_1, e_2\rangle : X \times Y}\,(\times_{\mathrm{I}}) \qquad \frac{\Gamma \vdash e : X \times Y}{\Gamma \vdash \text{fst } e : X}\,(\times_{\mathrm{E1}}) \qquad \frac{\Gamma \vdash e : X \times Y}{\Gamma \vdash \text{snd } e : Y}\,(\times_{\mathrm{E2}})$$

$$\frac{\Gamma \vdash e : X}{\Gamma \vdash \mathrm{L}\ e : X + Y}\,(+_{\mathrm{I1}}) \qquad \frac{\Gamma \vdash e : Y}{\Gamma \vdash \mathrm{R}\ e : X + Y}\,(+_{\mathrm{I2}})$$

$$\frac{\Gamma \vdash e : X + Y \quad \Gamma, x : X \vdash e_1 : Z \quad \Gamma, y : Y \vdash e_2 : Z}{\Gamma \vdash \text{case } (e, \mathrm{L}\ x \to e_1, \mathrm{R}\ y \to e_2) : Z}\,(+_{\mathrm{E}})$$

The typed lambda calculi are total: all terms terminate as the types prevent the construction of a term like Ω.

Intuitionistic Propositional Logic

STLC is equivalent to IPL.

$$\frac{P \in \Psi}{\Psi \vdash P\ \text{true}} \qquad \frac{\Psi \vdash \bot\ \text{true}}{\Psi \vdash R\ \text{true}} \qquad \frac{}{\Psi \vdash \top\ \text{true}}$$

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

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

$$\frac{\Psi, P \vdash Q\ \text{true}}{\Psi \vdash P \Longrightarrow Q\ \text{true}} \qquad \frac{\Psi \vdash P \Longrightarrow Q\ \text{true} \quad \Psi \vdash P\ \text{true}}{\Psi \vdash Q\ \text{true}}$$

Negation is **derived**: $\neg P \stackrel{\text{def}}{=} P \Longrightarrow \bot$. Refuting P means proving that $P \Longrightarrow \bot$.

Intuitionistic because there's no law of the excluded middle: $\neg P \vee P$ does not hold, nor does $\neg\neg P \Longrightarrow P$.

Polymorphic Lambda Calculus/System F

Types	$A ::= \alpha \mid A \to A \mid \forall \alpha.\ A$
Terms	$e\ ::= x \mid \lambda x : X.\ e \mid e\ e \mid \Lambda \alpha.\ e \mid e\ \alpha$
Type Contexts	$\Theta ::= \cdot \mid \Theta, \alpha$
Term Variable Contexts	$\Gamma\ ::= \cdot \mid \Gamma, x : X$

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

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

Types can have free variables, so we need to ensure types are well-scoped ($\Theta \vdash A$ type). We then need judgements to ensure that type contexts are well-formed ($\Theta \vdash \Gamma$ ctx) as they contain types, which have a well-formedness condition.

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

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

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

Important that whenever we're dealing with a type or a context, we have an appropriate well-formedness condition around: all theorems using A should also include $\Theta \vdash A$ type, etc.

Second-Order Intuitionistic Logic

PLC is equivalent to SIL: but we don't know SIL, so that's... great.

15 Proofs

Progress: let bindings

If $\cdot \vdash e : \tau$ then either e is a value or $\exists e'.\ e \rightsquigarrow e'$.

By structural induction on $\vdash$:

...

Case $\dfrac{\Gamma \vdash e_1 : t' \quad \Gamma, x : t' \vdash e_2 : t}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : t}\,(\text{let})$:

Assume $\cdot \vdash e_1 : t'$ and $x : t' \vdash e_2 : t$ from the top of the rule. Progress holds for them as they're subderivations.

RTP Progress, so assume $\cdot \vdash \text{let } x = e_1 \text{ in } e_2 : t$ and need to prove that the let-binding is either a value or that it steps.

Using Progress on e_1, either e_1 is a value or $\exists e_1'.\ e_1 \rightsquigarrow e_1'$.

Case e_1 is a value:

By $\dfrac{}{\text{let } x = e_1 \text{ in } e_2 \rightsquigarrow e_2[e_1/x]}\,(\text{op-let-1})$, $\text{let } x = e_1 \text{ in } e_2 \rightsquigarrow e_2[e_1/x]$.

Case $\exists e_1'.\ e_1 \rightsquigarrow e_1'$:

By $\dfrac{e_1 \rightsquigarrow e_1'}{\text{let } x = e_1 \text{ in } e_2 \rightsquigarrow \text{let } x = e_1' \text{ in } e_2}\,(\text{op-let-2})$, let $x = e_1$ in $e_2 \rightsquigarrow$ let $x = e_1'$ in e_2.

...