Polymorphism Example

Consider a “doubling” function that takes a function *f*, and an integer *x*, applies *f* to *x*, and then applies *f* to the result:

Polymorphism Example

Consider a "doubling" function that takes a function f, and an integer x, applies f to x, and then applies f to the result:

$$\text{doubleInt} \triangleq \lambda f\!:\!\mathbf{int} \rightarrow \mathbf{int}.\ \lambda x\!:\!\mathbf{int}.\ f\,(f\,x)$$

Polymorphism Example

Consider a "doubling" function that takes a function *f*, and an integer *x*, applies *f* to *x*, and then applies *f* to the result:

$$\text{doubleInt} \triangleq \lambda f\!:\mathbf{int} \to \mathbf{int}.\ \lambda x\!:\mathbf{int}.\ f\,(f\,x)$$

Now suppose we want the same function for Booleans, or functions...

$$\begin{aligned} \text{doubleBool} &\triangleq \lambda f\!:\mathbf{bool} \to \mathbf{bool}.\ \lambda x\!:\mathbf{bool}.\ f\,(f\,x) \\ \text{doubleFn} &\triangleq \lambda f\!:(\mathbf{int} \to \mathbf{int}) \to (\mathbf{int} \to \mathbf{int}).\ \lambda x\!:\mathbf{int} \to \mathbf{int}.\ f\,(f\,x) \\ &\vdots \end{aligned}$$

Abstraction

These examples on the preceding slides violate a fundamental principle of software engineering:

Definition (Abstraction Principle)

Every major piece of functionality in a program should be implemented in just one place in the code. When similar functionality is provided by distinct pieces of code, the two should be combined into one by abstracting out the varying parts.

Polymorphic λ-Calculus

Invented independently in 1972–1974 by a computer scientist John Reynolds and a logician Jean-Yves Girard (who called it System F).

Key feature: Function abstraction and application, just like in λ-calculus terms, but *at the type level!*

Notation:

- $\Lambda\alpha.\, e$: type abstraction
- $e[\tau]$: type application

Example:
$\Lambda\alpha.\, \lambda x\!:\!\alpha.\, x$

Polymorphic λ-Calculus

Syntax

$$
\begin{aligned}
e &::= n \mid x \mid \lambda x\!:\!\tau.\, e \mid e_1\, e_2 \\
v &::= n \mid \lambda x\!:\!\tau.\, e
\end{aligned}
$$

Polymorphic λ-Calculus

Syntax

$$
\begin{aligned}
e &::= n \mid x \mid \lambda x\!:\!\tau.\, e \mid e_1\, e_2 \mid \Lambda\alpha.\, e \\
v &::= n \mid \lambda x\!:\!\tau.\, e
\end{aligned}
$$

Polymorphic λ-Calculus

Syntax

$$e ::= n \mid x \mid \lambda x\!:\!\tau.\, e \mid e_1\, e_2 \mid \Lambda\alpha.\, e \mid e\,[\tau]$$

$$v ::= n \mid \lambda x\!:\!\tau.\, e$$

Polymorphic λ-Calculus

Syntax

$$e ::= n \mid x \mid \lambda x\!:\!\tau.\, e \mid e_1\, e_2 \mid \Lambda\alpha.\, e \mid e\,[\tau]$$

$$v ::= n \mid \lambda x\!:\!\tau.\, e \mid \Lambda\alpha.\, e$$

Polymorphic λ-Calculus

Syntax

$$e ::= n \mid x \mid \lambda x\!:\!\tau.\, e \mid e_1\, e_2 \mid \Lambda\alpha.\, e \mid e\,[\tau]$$

$$v ::= n \mid \lambda x\!:\!\tau.\, e \mid \Lambda\alpha.\, e$$

Dynamic Semantics

$$\frac{}{(\lambda x\!:\!\tau.\, e)\, v \to e\{x \mapsto v\}}$$

Polymorphic λ-Calculus

Syntax

$$e ::= n \mid x \mid \lambda x\!:\!\tau.\, e \mid e_1\, e_2 \mid \Lambda\alpha.\, e \mid e\, [\tau]$$
$$v ::= n \mid \lambda x\!:\!\tau.\, e \mid \Lambda\alpha.\, e$$

Dynamic Semantics

$$\frac{}{(\lambda x\!:\!\tau.\, e)\, v \to e\{x \mapsto v\}} \qquad \frac{}{(\Lambda\alpha.\, e)\, [\tau] \to e\{\alpha \mapsto \tau\}}$$

Typing Judgment

Type Syntax

$$\alpha \in \textbf{TVar}$$
$$\tau ::= \textbf{int} \mid \tau_1 \to \tau_2$$

Typing Judgment

Type Syntax

$$\alpha \in \textbf{TVar}$$
$$\tau ::= \textbf{int} \mid \tau_1 \rightarrow \tau_2 \mid \alpha$$

Typing Judgment

Type Syntax

$$\alpha \in \textbf{TVar}$$
$$\tau ::= \textbf{int} \mid \tau_1 \to \tau_2 \mid \alpha \mid \forall \alpha.\, \tau$$

Typing Judgment

Type Syntax

$$\alpha \in \textbf{TVar}$$
$$\tau ::= \textbf{int} \mid \tau_1 \to \tau_2 \mid \alpha \mid \forall \alpha.\, \tau$$

Typing Judgment: $\Delta, \Gamma \vdash e : \tau$

- Γ a mapping from variables to types
- Δ a set of type variables in scope

Typing Judgment

Type Syntax

$$\alpha \in \textbf{TVar}$$
$$\tau ::= \textbf{int} \mid \tau_1 \to \tau_2 \mid \alpha \mid \forall \alpha.\, \tau$$

Typing Judgment: $\Delta, \Gamma \vdash e : \tau$

- Γ a mapping from variables to types
- Δ a set of type variables in scope

Type Well-Formedness: $\Delta \vdash \tau$ ok

For example, $\alpha \to$ **int** is valid type syntax, but it is not well-formed. But $\forall \alpha.\, \alpha \to$ **int** is.

Typing Judgment

Type Syntax

$$\alpha \in \textbf{TVar}$$
$$\tau ::= \textbf{int} \mid \tau_1 \rightarrow \tau_2 \mid \alpha \mid \forall\alpha.\ \tau$$

Typing Judgment: $\Delta, \Gamma \vdash e : \tau$

- Γ a mapping from variables to types
- Δ a set of type variables in scope

Type Well-Formedness: $\Delta \vdash \tau$ ok

For example, $\alpha \rightarrow \textbf{int}$ is valid type syntax, but it is not well-formed. But $\forall\alpha.\ \alpha \rightarrow \textbf{int}$ is.

$$(\Lambda\alpha.\ \lambda a : \alpha.\ 42)$$

Typing Judgment

Type Syntax

$$\alpha \in \textbf{TVar}$$

$$\tau ::= \textbf{int} \mid \tau_1 \to \tau_2 \mid \alpha \mid \forall \alpha.\ \tau$$

Typing Judgment: $\Delta, \Gamma \vdash e : \tau$

- Γ a mapping from variables to types
- Δ a set of type variables in scope

Type Well-Formedness: $\Delta \vdash \tau$ ok

For example, $\alpha \to \textbf{int}$ is valid type syntax, but it is not well-formed. But $\forall \alpha.\ \alpha \to \textbf{int}$ is.

$$\{\}, \{\} \vdash (\Lambda \alpha.\ \lambda a : \alpha.\ 42) : \forall \alpha.\ \alpha \to \textbf{int}$$

Typing Rules

$$\frac{}{\Delta, \Gamma \vdash n : \textbf{int}}$$

Typing Rules

$$\frac{}{\Delta, \Gamma \vdash n : \textbf{int}} \qquad \frac{\Gamma(x) = \tau}{\Delta, \Gamma \vdash x : \tau}$$

Typing Rules

$$\frac{}{\Delta, \Gamma \vdash n : \textbf{int}}$$

$$\frac{\Gamma(x) = \tau}{\Delta, \Gamma \vdash x : \tau}$$

$$\frac{\Delta, \Gamma, x : \tau \vdash e : \tau' \quad \Delta \vdash \tau \text{ ok}}{\Delta, \Gamma \vdash \lambda x : \tau.\, e : \tau \rightarrow \tau'}$$

Typing Rules

$$\frac{}{\Delta, \Gamma \vdash n : \mathbf{int}} \qquad \frac{\Gamma(x) = \tau}{\Delta, \Gamma \vdash x : \tau}$$

$$\frac{\Delta, \Gamma, x : \tau \vdash e : \tau' \quad \Delta \vdash \tau \text{ ok}}{\Delta, \Gamma \vdash \lambda x : \tau.\, e : \tau \to \tau'} \qquad \frac{\Delta, \Gamma \vdash e_1 : \tau \to \tau' \quad \Delta, \Gamma \vdash e_2 : \tau}{\Delta, \Gamma \vdash e_1\, e_2 : \tau'}$$

Typing Rules

$$\frac{}{\Delta, \Gamma \vdash n : \mathbf{int}} \qquad \frac{\Gamma(x) = \tau}{\Delta, \Gamma \vdash x : \tau}$$

$$\frac{\Delta, \Gamma, x : \tau \vdash e : \tau' \quad \Delta \vdash \tau \text{ ok}}{\Delta, \Gamma \vdash \lambda x : \tau.\, e : \tau \to \tau'} \qquad \frac{\Delta, \Gamma \vdash e_1 : \tau \to \tau' \quad \Delta, \Gamma \vdash e_2 : \tau}{\Delta, \Gamma \vdash e_1\, e_2 : \tau'}$$

$$\frac{\Delta \cup \{\alpha\}, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\, \tau}$$

Typing Rules

$$\frac{}{\Delta, \Gamma \vdash n : \textbf{int}} \qquad \frac{\Gamma(x) = \tau}{\Delta, \Gamma \vdash x : \tau}$$

$$\frac{\Delta, \Gamma, x : \tau \vdash e : \tau' \quad \Delta \vdash \tau \text{ ok}}{\Delta, \Gamma \vdash \lambda x : \tau.\, e : \tau \to \tau'} \qquad \frac{\Delta, \Gamma \vdash e_1 : \tau \to \tau' \quad \Delta, \Gamma \vdash e_2 : \tau}{\Delta, \Gamma \vdash e_1\, e_2 : \tau'}$$

$$\frac{\Delta \cup \{\alpha\}, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\, \tau} \qquad \frac{\Delta, \Gamma \vdash e : \forall\alpha.\, \tau' \quad \Delta \vdash \tau \text{ ok}}{\Delta, \Gamma \vdash e\, [\tau] : \tau'\{\alpha \mapsto \tau\}}$$

Type Well-Formedness

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha \ \mathsf{ok}}$$

Type Well-Formedness

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha \ \mathsf{ok}}$$

$$\frac{}{\Delta \vdash \textbf{int} \ \mathsf{ok}} \qquad \frac{}{\Delta \vdash \textbf{bool} \ \mathsf{ok}}$$

Type Well-Formedness

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha \ \mathsf{ok}}$$

$$\frac{}{\Delta \vdash \textbf{int} \ \mathsf{ok}} \qquad \frac{}{\Delta \vdash \textbf{bool} \ \mathsf{ok}}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{ok} \quad \Delta \vdash \tau_2 \ \mathsf{ok}}{\Delta \vdash \tau_1 \to \tau_2 \ \mathsf{ok}}$$

Type Well-Formedness

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha \; \mathsf{ok}}$$

$$\frac{}{\Delta \vdash \textbf{int} \; \mathsf{ok}} \qquad \frac{}{\Delta \vdash \textbf{bool} \; \mathsf{ok}}$$

$$\frac{\Delta \vdash \tau_1 \; \mathsf{ok} \quad \Delta \vdash \tau_2 \; \mathsf{ok}}{\Delta \vdash \tau_1 \rightarrow \tau_2 \; \mathsf{ok}}$$

$$\frac{\Delta \cup \{\alpha\} \vdash \tau \; \mathsf{ok}}{\Delta \vdash \forall \alpha.\, \tau \; \mathsf{ok}}$$

Example: Doubling Redux

Let's consider the doubling operation again.

Example: Doubling Redux

Let's consider the doubling operation again.

We can write a polymorphic doubling operation as:

$$\text{double} \triangleq \Lambda\alpha.\, \lambda f{:}\,\alpha \to \alpha.\, \lambda x{:}\,\alpha.\, f\,(f\,x)$$

Example: Doubling Redux

Let's consider the doubling operation again.

We can write a polymorphic doubling operation as:

$$\text{double} \triangleq \Lambda\alpha.\, \lambda f{:}\,\alpha \to \alpha.\, \lambda x{:}\,\alpha.\, f\,(f\,x)$$

The type of this expression is: $\forall\alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$

Example: Doubling Redux

Let's consider the doubling operation again.

We can write a polymorphic doubling operation as:

$$\text{double} \triangleq \Lambda\alpha.\, \lambda f{:}\,\alpha \to \alpha.\, \lambda x{:}\,\alpha.\, f\,(f\,x)$$

The type of this expression is: $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$

We can instantiate this on a type, and provide arguments:

Example: Doubling Redux

Let's consider the doubling operation again.

We can write a polymorphic doubling operation as:

$$\text{double} \triangleq \Lambda\alpha.\, \lambda f{:}\,\alpha \to \alpha.\, \lambda x{:}\,\alpha.\, f\,(f\,x)$$

The type of this expression is: $\forall\alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$

We can instantiate this on a type, and provide arguments:

$$\text{double}\ [\textbf{int}]\ (\lambda n{:}\,\textbf{int}.\, n + 1)\ 7$$

Example: Doubling Redux

Let's consider the doubling operation again.

We can write a polymorphic doubling operation as:

$$\text{double} \triangleq \Lambda\alpha.\, \lambda f{:}\,\alpha \to \alpha.\, \lambda x{:}\,\alpha.\, f\,(f\,x)$$

The type of this expression is: $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$

We can instantiate this on a type, and provide arguments:

$$\begin{array}{rl} & \text{double}\,[\textbf{int}]\,(\lambda n : \textbf{int}.\, n + 1)\, 7 \\ \to & (\lambda f{:}\,\textbf{int} \to \textbf{int}.\, \lambda x{:}\,\textbf{int}.\, f\,(f\,x))\,(\lambda n : \textbf{int}.\, n + 1)\, 7 \end{array}$$

Example: Doubling Redux

Let's consider the doubling operation again.

We can write a polymorphic doubling operation as:

$$\text{double} \triangleq \Lambda\alpha.\, \lambda f\!:\alpha \to \alpha.\, \lambda x\!:\alpha.\, f\,(f\,x)$$

The type of this expression is: $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$

We can instantiate this on a type, and provide arguments:

$$\begin{array}{ll} & \text{double}\,[\mathbf{int}]\,(\lambda n\!:\mathbf{int}.\, n+1)\;7 \\ \to & (\lambda f\!:\mathbf{int} \to \mathbf{int}.\, \lambda x\!:\mathbf{int}.\, f\,(f\,x))\,(\lambda n\!:\mathbf{int}.\, n+1)\;7 \\ \to^* & 9 \end{array}$$

Inference Rules for Logic

A seeming non sequitur: let's use inference rules to define a logical system.

Inference Rules for Logic

A seeming non sequitur: let's use inference rules to define a logical system.

Here's a rule from natural deduction, a *constructive* logic invented by logician Gerhard Gentzen in 1935:

$$\frac{\phi \qquad \psi}{\phi \wedge \psi} \;\wedge\text{-INTRO}$$

Given a proof of ϕ and a proof of ψ, the rule lets you *construct* a proof of $\phi \wedge \psi$.

Natural Deduction

Let's use our usual PL tools to define the set of true formulas ("theorems").

Natural Deduction

Let's use our usual PL tools to define the set of true formulas ("theorems").

We'll start with a grammar for formulas:

$$
\begin{array}{rcl}
\phi & ::= & \top \\
& \mid & \bot \\
& \mid & X \\
& \mid & \phi \wedge \psi \\
& \mid & \phi \vee \psi \\
& \mid & \phi \rightarrow \psi \\
& \mid & \neg\phi \\
& \mid & \forall X.\ \phi
\end{array}
$$

where X ranges over Boolean variables
and $\neg\phi$ is an abbreviation for $\phi \rightarrow \bot$.

Natural Deduction

Let's define a judgment that that a formula is true given a set of assumptions Γ:

$$\Gamma \vdash \phi$$

where Γ is just a list of formulas.

Natural Deduction

Let's define a judgment that that a formula is true given a set of assumptions Γ:

$$\Gamma \vdash \phi$$

where Γ is just a list of formulas.

If $\vdash \phi$ (with no assumptions), we say ϕ is a *theorem*.

Examples:

- $\vdash A \land B \rightarrow A$

Natural Deduction

Let's define a judgment that that a formula is true given a set of assumptions Γ:

$$\Gamma \vdash \phi$$

where Γ is just a list of formulas.

If $\vdash \phi$ (with no assumptions), we say ϕ is a *theorem*.

Examples:

- $\vdash A \land B \to A$
- $\vdash \neg(A \land B) \to \neg A \lor \neg B$

Natural Deduction

Let's define a judgment that that a formula is true given a set of assumptions Γ:

$$\Gamma \vdash \phi$$

where Γ is just a list of formulas.

If $\vdash \phi$ (with no assumptions), we say ϕ is a *theorem*.

Examples:

- $\vdash A \land B \to A$
- $\vdash \neg(A \land B) \to \neg A \lor \neg B$
- $A, B, C \vdash B$

Natural Deduction

Let's write the rules for our judgment:

$$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \land \psi} \land\text{-INTRO}$$

Natural Deduction

Let's write the rules for our judgment:

$$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi} \wedge\text{-INTRO}$$

$$\frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi} \wedge\text{-ELIM1} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi} \wedge\text{-ELIM2}$$

Natural Deduction

Let's write the rules for our judgment:

$$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi}\ \wedge\text{-INTRO}$$

$$\frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi}\ \wedge\text{-ELIM1} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi}\ \wedge\text{-ELIM2}$$

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi}\ \to\text{-INTRO}$$

Natural Deduction

Let's write the rules for our judgment:

$$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \land \psi} \land\text{-INTRO}$$

$$\frac{\Gamma \vdash \phi \land \psi}{\Gamma \vdash \phi} \land\text{-ELIM1} \qquad \frac{\Gamma \vdash \phi \land \psi}{\Gamma \vdash \psi} \land\text{-ELIM2}$$

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi} \to\text{-INTRO}$$

...and so on.

Natural Deduction

$$\frac{}{\Gamma, \phi \vdash \phi}\ \textsc{Axiom}$$

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi}\ \to\text{-INTRO} \qquad \frac{\Gamma \vdash \phi \to \psi \qquad \Gamma \vdash \phi}{\Gamma \vdash \psi}\ \to\text{-ELIM}$$

$$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \land \psi}\ \bigwedge\text{-INTRO} \qquad \frac{\Gamma \vdash \phi \land \psi}{\Gamma \vdash \phi}\ \bigwedge\text{-ELIM1} \qquad \frac{\Gamma \vdash \phi \land \psi}{\Gamma \vdash \psi}\ \bigwedge\text{-ELIM2}$$

$$\frac{\Gamma \vdash \phi}{\Gamma \vdash \phi \lor \psi}\ \bigvee\text{-INTRO1} \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \phi \lor \psi}\ \bigvee\text{-INTRO2}$$

$$\frac{\Gamma \vdash \phi \lor \psi \qquad \Gamma \vdash \phi \to \chi \qquad \Gamma \vdash \psi \to \chi}{\Gamma \vdash \chi}\ \bigvee\text{-ELIM}$$

$$\frac{\Gamma, P \vdash \phi}{\Gamma \vdash \forall P.\, \phi}\ \forall\text{-INTRO} \qquad \frac{\Gamma \vdash \forall P.\, \phi}{\Gamma \vdash \phi\{\psi / P\}}\ \forall\text{-ELIM}$$

Natural Deduction

Let's try a proof! We can write a proof that $A \land B \to B \land A$ is a theorem.

Natural Deduction

Let's try a proof! We can write a proof that $A \wedge B \rightarrow B \wedge A$ is a theorem.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{A \wedge B \vdash A \wedge B}\ \textsf{AXIOM}}{A \wedge B \vdash B}\ \wedge\textsf{-ELIM2} \qquad \cfrac{\cfrac{}{A \wedge B \vdash A \wedge B}\ \textsf{AXIOM}}{A \wedge B \vdash A}\ \wedge\textsf{-ELIM1}}{A \wedge B \vdash B \wedge A}\ \wedge\textsf{-INTRO}}{\vdash A \wedge B \rightarrow B \wedge A}\ \rightarrow\textsf{-INTRO}$$

Natural Deduction

Let's try a proof! We can write a proof that $A \land B \to B \land A$ is a theorem.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \land B \vdash A \land B}\ \textsf{AXIOM}}{A \land B \vdash B}\ \land\textsf{-ELIM2} \qquad \dfrac{\dfrac{}{A \land B \vdash A \land B}\ \textsf{AXIOM}}{A \land B \vdash A}\ \land\textsf{-ELIM1}}{A \land B \vdash B \land A}\ \land\textsf{-INTRO}}{\vdash A \land B \to B \land A}\ \to\textsf{-INTRO}$$

Does this look familiar?

Natural Deduction

Let's try a proof! We can write a proof that $A \land B \to B \land A$ is a theorem.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \land B \vdash A \land B}\ \text{AXIOM}}{A \land B \vdash B}\ \land\text{-ELIM2} \qquad \dfrac{\dfrac{}{A \land B \vdash A \land B}\ \text{AXIOM}}{A \land B \vdash A}\ \land\text{-ELIM1}}{A \land B \vdash B \land A}\ \land\text{-INTRO}}{\vdash A \land B \to B \land A}\ \to\text{-INTRO}$$

Does this look familiar?

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}A \times B \vdash x{:}A \times B}\ \text{T-VAR}}{x{:}A \times B \vdash \#2\ x{:}B}\ \text{T-\#1} \qquad \dfrac{\dfrac{}{x{:}A \times B \vdash x{:}A \times B}\ \text{T-VAR}}{x{:}A \times B \vdash \#1\ x{:}A}\ \text{T-\#2}}{x{:}A \times B \vdash (\#2\ x, \#1\ x){:}B \times A}\ \text{T-PAIR}}{\vdash \lambda x.\,(\#2\ x, \#1\ x){:}A \times B \to B \times A}\ \text{T-ABS}$$

Propositions as Types

Every natural deduction proof tree has a corresponding type tree in System F with product and sum types! And vice-versa!

Type Systems		Formal Logic	
τ	Type	ϕ	Formula
τ	is inhabited	ϕ	is a theorem
e	Well-typed expression	π	Proof

A program with a given type acts as a *witness* that the type's corresponding formula is true.

Propositions as Types

Every type rule in System F with product and sum types corresponds 1-1 with a proof rule in natural deduction:

Type Systems		**Formal Logic**	
$\rightarrow$	Function	$\rightarrow$	Implication
$\times$	Product	$\wedge$	Conjunction
$+$	Sum	$\vee$	Disjunction
$\forall$	Universal	$\forall$	Quantifier

You can even add existential types to correspond to existential quantification. It still works!

Propositions as Types

Every type rule in System F with product and sum types corresponds 1-1 with a proof rule in natural deduction:

Type Systems		**Formal Logic**	
$\rightarrow$	Function	$\rightarrow$	Implication
$\times$	Product	$\wedge$	Conjunction
$+$	Sum	$\vee$	Disjunction
$\forall$	Universal	$\forall$	Quantifier

You can even add existential types to correspond to existential quantification. It still works!

Is this a coincidence? Natural deduction was invented by a German logician in 1935. Types for the λ-calculus were invented by Church at Princeton in 1940.

Propositions as Types Through the Ages

Natural Deduction Gentzen (1935)	$\Leftrightarrow$	**Typed λ-Calculus** Church (1940)
Type Schemes Hindley (1969)	$\Leftrightarrow$	**ML's Type System** Milner (1975)
System F Girard (1972)	$\Leftrightarrow$	**Polymorphic λ-Calculus** Reynolds (1974)
Modal Logic Lewis (1910)	$\Leftrightarrow$	**Monads** Kleisli (1965), Moggi (1987)
Classical–Intuitionistic Embedding Gödel (1933)	$\Leftrightarrow$	**Continuation Passing Style** Reynolds (1972)

Term Assignment

This all means that we have a new way of proving theorems: writing programs!

Term Assignment

This all means that we have a new way of proving theorems: writing programs!

To prove a formula ϕ:

1. Convert the ϕ into its corresponding type τ.
2. Find some program *e* that has the type τ.
3. Realize that the existence of *v* implies a type tree for $\vdash e : \tau$, which implies a proof tree for $\vdash \phi$.

Linear Logic

Linear logic is a very different kind of logic, introduced by Jean-Yves Girard in 1987 (in the Curry–Howard era).

Linear Logic

Linear logic is a very different kind of logic, introduced by Jean-Yves Girard in 1987 (in the Curry–Howard era).

"Normal" logic is meant to represent truth. And facts stay true even after to *use* them to prove other facts:

$$A \to B, A \to C, A \vdash B \wedge C$$

Linear Logic

In linear logic, a better intuition is *the conservation of matter*, as in a chemical reaction. We can't reuse A twice:

$$A \multimap B, A \multimap C, A \nvdash B \otimes C$$

(Where $\multimap$ is matter-preserving implication, and $\otimes$ is like $\wedge$ but for linear resources.)

Linear Logic

In linear logic, a better intuition is *the conservation of matter*, as in a chemical reaction. We can't reuse A twice:

$$A \multimap B, A \multimap C, A \nvdash B \otimes C$$

(Where $\multimap$ is matter-preserving implication, and $\otimes$ is like $\wedge$ but for linear resources.)

You would need two copies of A:

$$A \multimap B, A \multimap C, A, A \vdash B \otimes C$$

Linear Logic Syntax

Here's a complete language for linear logic formulas:

$$\phi ::= A \mid \phi \multimap \psi \mid \phi \otimes \psi \mid \phi \oplus \psi$$

where $\multimap$ is like an intuitionistic $\rightarrow$, $\otimes$ is like $\wedge$, and $\oplus$ is like $\vee$.

Linear Logic Inference Rules

Compare the intuitionistic rule for $\wedge$ introduction with the linear rule for $\otimes$ introduction:

$$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi}\ \wedge\text{-INTRO} \qquad\qquad \frac{\Gamma_1 \vdash \phi \qquad \Gamma_2 \vdash \psi}{\Gamma_1, \Gamma_2 \vdash \phi \otimes \psi}\ \otimes\text{-INTRO}$$

Contexts Γ are now like lists, not sets!

Linear Logic Inference Rules

$$\frac{}{\phi \vdash \phi}\ \textsf{AXIOM} \qquad \frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \multimap \psi}\ \multimap\textsf{-INTRO}$$

$$\frac{\Gamma_1 \vdash \phi \multimap \psi \qquad \Gamma_2 \vdash \phi}{\Gamma_1, \Gamma_2 \vdash \psi}\ \multimap\textsf{-ELIM} \qquad \frac{\Gamma_1 \vdash \phi \qquad \Gamma_2 \vdash \psi}{\Gamma_1, \Gamma_2 \vdash \phi \otimes \psi}\ \otimes\textsf{-INTRO}$$

$$\frac{\Gamma_1 \vdash \phi \otimes \psi \qquad \Gamma_2, \phi, \psi \vdash \chi}{\Gamma_1, \Gamma_2 \vdash \chi}\ \otimes\textsf{-ELIM} \qquad \frac{\Gamma \vdash \phi}{\Gamma \vdash \phi \oplus \psi}\ \oplus\textsf{-INTRO-L}$$

$$\frac{\Gamma \vdash \psi}{\Gamma \vdash \phi \oplus \psi}\ \oplus\textsf{-INTRO-R}$$

$$\frac{\Gamma_1 \vdash \phi \oplus \psi \qquad \Gamma_2, \phi \vdash \chi \qquad \Gamma_2, \psi \vdash \chi}{\Gamma_1, \Gamma_2 \vdash \chi}\ \oplus\textsf{-ELIM}$$

The Structural Rules

In an intuitionistic world, these rules are so boring that we don't usually even write them down. But they're critical for highlighting the difference with linear logic:

$$\frac{\Gamma \vdash \phi}{\Gamma, \psi \vdash \phi}\ \text{WEAKENING} \qquad \frac{\Gamma_1, \Gamma_2 \vdash \phi}{\Gamma_2, \Gamma_1 \vdash \phi}\ \text{EXCHANGE}$$

$$\frac{\Gamma, \psi, \psi \vdash \phi}{\Gamma, \psi \vdash \phi}\ \text{CONTRACTION}$$

The Structural Rules

$$\frac{\Gamma \vdash \phi}{\Gamma, \psi \vdash \phi}\ \text{WEAKENING} \qquad \frac{\Gamma_1, \Gamma_2 \vdash \phi}{\Gamma_2, \Gamma_1 \vdash \phi}\ \text{EXCHANGE}$$

$$\frac{\Gamma, \psi, \psi \vdash \phi}{\Gamma, \psi \vdash \phi}\ \text{CONTRACTION}$$

Eliminating these rules produces a family of *substructural logics:*

- **Linear logic:** Exchange only. Matter may neither be created nor destroyed.
- **Affine logic:** Exchange & weakening. You can use things or ignore them, but not duplicate them.
- **Relevant logic:** Exchange & contraction. Use everything at least once.
- **Ordered logic:** None. Use everything exactly once, in order.

Substructural Type Systems

Via Curry–Howard, every substructural logic becomes a *substructural type system:*

- **Linear logic:** Use every variable exactly once!
- **Affine logic:** Use every variable *at most* once!
- **Relevant logic:** Use every variable *at least* once!
- **Ordered logic:** Use every variable once, in order??

Substructural Type Systems

Via Curry–Howard, every substructural logic becomes a *substructural type system:*

- **Linear logic:** Use every variable exactly once!
- **Affine logic:** Use every variable *at most* once!
- **Relevant logic:** Use every variable *at least* once!
- **Ordered logic:** Use every variable once, in order??

$$\vdash (\lambda x : \mathbf{int}.\, x + x) : \mathbf{int} \to \mathbf{int}$$

Substructural Type Systems

Via Curry–Howard, every substructural logic becomes a *substructural type system:*

- **Linear logic:** Use every variable exactly once!
- **Affine logic:** Use every variable *at most* once!
- **Relevant logic:** Use every variable *at least* once!
- **Ordered logic:** Use every variable once, in order??

$$\vdash (\lambda x{:}\mathbf{int}.\, x + x) : \mathbf{int} \to \mathbf{int} \qquad \nvdash (\lambda x{:}\mathbf{int}.\, x + x) : \mathbf{int} \multimap \mathbf{int}$$

Applications of Substructural Types

Imagine a language with pointers. You can allocate memory, load and store through pointers, and free memory:

$$\begin{aligned}&\text{let } p \ = \ \text{malloc } 4 \text{ in}\\&\text{store } p \ ((\text{load } p) + 38);\\&\text{free } p\end{aligned}$$

Applications of Substructural Types

Imagine a language with pointers. You can allocate memory, load and store through pointers, and free memory:

$$\begin{aligned} &\text{let } p = \text{malloc } 4 \text{ in} \\ &\text{store } p\ ((\text{load } p) + 38); \\ &\text{free } p \end{aligned}$$

Everyone who has ever written C has written a *double-free* bug:

$$\begin{aligned} &\text{let } p = \text{malloc } 4 \text{ in} \\ &\text{free } p; \\ &\text{free } p \end{aligned}$$

Applications of Substructural Types

The unsafe (C-like) load and store "functions" have these types:

$$\mathsf{store} : \forall \alpha.\ (\alpha\ \mathsf{ptr} \times \alpha) \to \mathsf{void} \qquad\qquad \mathsf{load} : \forall \alpha.\ \alpha\ \mathsf{ptr} \to \alpha$$

Applications of Substructural Types

The unsafe (C-like) load and store "functions" have these types:

$$\mathsf{store} : \forall\alpha.\ (\alpha\ \mathsf{ptr} \times \alpha) \rightarrow \mathsf{void} \qquad \mathsf{load} : \forall\alpha.\ \alpha\ \mathsf{ptr} \rightarrow \alpha$$

The obvious linear versions are too restrictive:

$$\mathsf{store} : \forall\alpha.\ (\alpha\ \mathsf{ptr} \times \alpha) \multimap \mathsf{void} \qquad \mathsf{load} : \forall\alpha.\ \alpha\ \mathsf{ptr} \multimap \alpha$$

Applications of Substructural Types

The unsafe (C-like) load and store "functions" have these types:

$$\text{store} : \forall\alpha.\ (\alpha\ \text{ptr} \times \alpha) \rightarrow \text{void} \qquad \text{load} : \forall\alpha.\ \alpha\ \text{ptr} \rightarrow \alpha$$

The obvious linear versions are too restrictive:

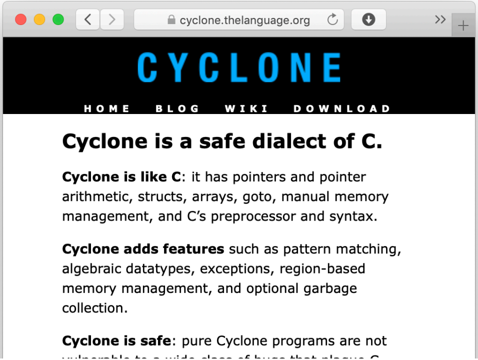
$$\text{store} : \forall\alpha.\ (\alpha\ \text{ptr} \times \alpha) \multimap \text{void} \qquad \text{load} : \forall\alpha.\ \alpha\ \text{ptr} \multimap \alpha$$

The trick is to "thread through" the pointer so you get a copy back on non-destructive operations:

$$\text{store} : \forall\alpha.\ (\alpha\ \text{ptr} \times \alpha) \multimap \alpha\ \text{ptr} \qquad \text{load} : \forall\alpha.\ \alpha\ \text{ptr} \multimap (\alpha\ \text{ptr} \times \alpha)$$

The destructive free function still consumes its argument and doesn't give it back.

Substructural Types for Memory Safety



Substructural Types for Memory Safety