Foundations of Software
Spring 2025

Week 09

Plan

PREVIOUSLY: Extensions to STLC (Pairs, Sums, Records, Recursion, State, etc.)

Plan

PREVIOUSLY: Extensions to STLC (Pairs, Sums, Records, Recursion, State, etc.)

TODAY: Polymorphism (System F)

1. Motivation: Code Reuse and Encapsulation
2. Intuitive Construction of System F
3. Formal Definition: Syntax, Typing, Reduction
4. Encodings in System F: Booleans, Naturals, Pairs, Encapsulation
5. Metatheory: Soundness, Normalization, Evaluation
6. Parametricity and Theorems for Free
7. Curry-Howard for System F
8. Other Kinds of Polymorphism

Motivation (1/2): Code Repetition

Consider writing common functions in $\lambda_{\rightarrow}$:

- `map` : (Int→Int)→List Int→List Int
- `map` : (Bool→Bool)→List Bool→List Bool
- `sort` : (Int→Int→Bool)→List Int→List Int

Problem: The core logic is identical, but $\lambda_{\rightarrow}$ forces us to write separate versions for each type. We cannot write a single generic `map` or `sort`.

Motivation (1/2): Code Repetition

Consider writing common functions in $\lambda_{\rightarrow}$:

- `map` : (Int→Int)→List Int→List Int
- `map` : (Bool→Bool)→List Bool→List Bool
- `sort` : (Int→Int→Bool)→List Int→List Int

Problem: The core logic is identical, but $\lambda_{\rightarrow}$ forces us to write separate versions for each type. We cannot write a single generic `map` or `sort`.

We want a way to parameterize functions by types.

Motivation (2/2): Encapsulation

Imagine providing a Stack data structure:

- We want to offer an interface:
  - `empty` : StackInt
  - `push` : Int→StackInt→StackInt
  - `pop` : StackInt→(Int × StackInt)
  - `isEmpty` : StackInt→Bool
- We want to *hide* the concrete implementation (e.g., using lists, arrays). Users should *only* interact through the interface types.

Motivation (2/2): Encapsulation

Imagine providing a Stack data structure:

- We want to offer an interface:
  - `empty` : StackInt
  - `push` : Int→StackInt→StackInt
  - `pop` : StackInt→(Int × StackInt)
  - `isEmpty` : StackInt→Bool
- We want to *hide* the concrete implementation (e.g., using lists, arrays). Users should *only* interact through the interface types.

Problem: $\lambda_{\rightarrow}$ doesn't directly support hiding implementation details behind an abstract type interface. We want machinery for abstract data types. Polymorphism (System F) will provide (partial) solutions for both code reuse and encapsulation.

Motivation (2/2): Encapsulation

Imagine providing a Stack data structure:

- We want to offer an interface:
  - `empty` : StackInt
  - `push` : Int→StackInt→StackInt
  - `pop` : StackInt→(Int × StackInt)
  - `isEmpty` : StackInt→Bool
- We want to *hide* the concrete implementation (e.g., using lists, arrays). Users should *only* interact through the interface types.

Problem: $\lambda_{\rightarrow}$ doesn't directly support hiding implementation details behind an abstract type interface. We want machinery for abstract data types. Polymorphism (System F) will provide (partial) solutions for both code reuse and encapsulation.

Bonus: Church encodings will be legal again!

Intuitive Construction of System F

Idea: Allow *types* themselves to be parameters.

Intuitive Construction of System F

Idea: Allow *types* themselves to be parameters.

Example: A generic ‘const‘ function: `const` 42, or `const` *true*

- We want to abstract over the return type of the domain, say β. Sometimes it will be int, or bool.
- Return a function that should itself accept different types!
  `const` $42\ 0 = 42$, `const` 42 *false* $= 42$

Intuitive Construction of System F

We introduce:

- Type variables (e.g., α, β) as placeholders for types.
- Type abstraction ($\Lambda\alpha.t$) to create functions that take a type as an argument.
- Type application ($t[\tau]$) to provide a concrete type to a polymorphic function, specializing it.

Intuitive Construction of System F

We introduce:

- Type variables (e.g., α, β) as placeholders for types.
- Type abstraction ($\Lambda\alpha.t$) to create functions that take a type as an argument.
- Type application ($t[\tau]$) to provide a concrete type to a polymorphic function, specializing it.

Example: Polymorphic constant function (‘const‘)

$$\texttt{const} = \Lambda\beta.\lambda x : \beta.\Lambda\alpha.\lambda y : \alpha.x$$

This function takes one type argument β and one term argument (x of type β), and returns a function that takes one type argument α and a term argument of type α and returns x.

Intuitive Construction of System F

We introduce:

- Type variables (e.g., α, β) as placeholders for types.
- Type abstraction ($\Lambda\alpha.t$) to create functions that take a type as an argument.
- Type application ($t[\tau]$) to provide a concrete type to a polymorphic function, specializing it.

Example: Polymorphic constant function ('const')

$$\texttt{const} = \Lambda\beta.\lambda x : \beta.\Lambda\alpha.\lambda y : \alpha.x$$

This function takes one type argument β and one term argument (x of type β), and returns a function that takes one type argument α and a term argument of type α and returns x.

The type of `const` is: $\forall\beta.\beta\rightarrow(\forall\alpha.\alpha\rightarrow\beta)$

Example in intuitive System F

$$\texttt{const} = \Lambda\beta.\lambda x : \beta.\Lambda\alpha.\lambda y : \alpha.x$$

Example in intuitive System F

$$\texttt{const} = \Lambda\beta.\lambda x : \beta.\Lambda\alpha.\lambda y : \alpha.x$$

Applying the function: We can specialize ‘const‘ by providing a first concrete type:

$$\texttt{const}[\mathsf{Int}]$$

This specialized function has type: $\mathsf{Int} \to \forall\beta, \beta \to \mathsf{Int}$
Then apply it to the value, then the other type and the value :

$$(\texttt{const}[\mathsf{Int}]\ 42)[\mathsf{Bool}]\ \textit{true}$$

This evaluates to 42.

$$(\texttt{const}[\mathsf{Int}]\ 42)[\mathsf{Int}]\ 13$$

This evaluates to 42.

System F: Formal Syntax

τ ::=		*Types*
	α	*Type variable*
	$\tau_1 \to \tau_2$	*Function type*
	$\forall \alpha . \tau$	*Universal type*
t ::=		*Terms*
	x	*Variable*
	$\lambda x : \tau . t$	*Lambda abstraction*
	$t_1 \; t_2$	*Application*
	$\Lambda \alpha . t$	*Type abstraction*
	$t[\tau]$	*Type application*
Γ ::=		*Contexts*
	$\emptyset$	*Empty context*
	$\Gamma, x : \tau$	*Term variable binding*
	Γ, α	*Type variable binding*

System F: Values

Values represent the results of computation. In System F, these are functions (waiting for a term argument) and polymorphic functions (waiting for a type argument).

v ::=		*Values*
	$\lambda x : \tau.t$	*Lambda abstraction value*
	$\lambda\alpha.t$	*Type abstraction value*

Note: Variables (x), applications ($t_1\ t_2$), and type applications ($t[\tau]$) are *not* values. Evaluation will proceed until one of the value forms above is reached.

System F: Typing Rules (1/2)

Variables and Abstraction/Application (like STLC, but context can have type variables):

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, x : \tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . t_2 : \tau_1 \rightarrow \tau_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : \tau_{11} \rightarrow \tau_{12} \qquad \Gamma \vdash t_2 : \tau_{11}}{\Gamma \vdash t_1\ t_2 : \tau_{12}} \qquad \text{(T-App)}$$

System F: Typing Rules (2/2)

New rules for Polymorphism:

$$\frac{\Gamma, \alpha \vdash t : \tau \qquad (\alpha \text{ not free in } \Gamma)}{\Gamma \vdash \Lambda\alpha.t : \forall\alpha.\tau} \qquad (\text{T-TAbs})$$

(*Introduces* a polymorphic type)

System F: Typing Rules (2/2)

New rules for Polymorphism:

$$\frac{\Gamma, \alpha \vdash t : \tau \qquad (\alpha \text{ not free in } \Gamma)}{\Gamma \vdash \Lambda\alpha.t : \forall\alpha.\tau} \qquad (\text{T-TAbs})$$

(*Introduces* a polymorphic type)

$$\frac{\Gamma \vdash t_1 : \forall\alpha.\tau_{11} \qquad \Gamma \vdash \tau_{12} \text{ type}}{\Gamma \vdash t_1[\tau_{12}] : [\alpha \mapsto \tau_{12}]\tau_{11}} \qquad (\text{T-TApp})$$

(*Eliminates* a polymorphic type by substitution)

System F: Reduction Rules 1/2

Standard Beta-reduction:

$$(\lambda x : \tau_1 . t_{12})\ v_2 \longrightarrow [\mathsf{x} \mapsto v_2] t_{12} \qquad \text{(E-AppAbs)}$$

System F: Reduction Rules 1/2

Standard Beta-reduction:

$$(\lambda x : \tau_1 . t_{12})\ v_2 \longrightarrow [x \mapsto v_2] t_{12} \qquad \text{(E-AppAbs)}$$

New reduction for Type Application:

$$(\Lambda \alpha . t_{11})[\tau_2] \longrightarrow [\alpha \mapsto \tau_2] t_{11} \qquad \text{(E-TAppTAbs)}$$

(Substitution happens in the term, not just the type)

System F: Reduction Rules (Congruence)

Congruence rules (standard for App1, App2, plus new one):

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-App2)}$$

$$\frac{t \longrightarrow t'}{t[\tau] \longrightarrow t'[\tau]} \qquad \text{(E-TApp)}$$

System F Example: Identity

Polymorphic Identity Function: $\mathsf{id} = \Lambda\alpha.\lambda x : \alpha.x$
Typing Derivation ($\Gamma = \emptyset$):

1. $\alpha, x : \alpha \vdash x : \alpha$ (T-Var)
2. $\alpha \vdash \lambda x : \alpha.x : \alpha \to \alpha$ (T-Abs)
3. $\emptyset \vdash \Lambda\alpha.\lambda x : \alpha.x : \forall\alpha.\alpha \to \alpha$ (T-TAbs)

Using the identity:

- $\mathsf{id}[\mathsf{Int}]$ has type $\mathsf{Int} \to \mathsf{Int}$ (T-TApp)
- $(\mathsf{id}[\mathsf{Int}])\ 5$ has type Int (T-App)

Reduction:

- $(\Lambda\alpha.\lambda x : \alpha.x)[\mathsf{Int}] \longrightarrow \lambda x : \mathsf{Int}.x$ (E-TAppTAbs)
- $(\lambda x : \mathsf{Int}.x)\ 5 \longrightarrow [\mathsf{x} \mapsto 5](x) = 5$ (E-AppAbs)

System F Example: Function Composition

Polymorphic Function Composition:

$$\text{compose} = \Lambda\alpha.\Lambda\beta.\Lambda\gamma.\lambda f : (\beta\to\gamma).\lambda g : (\alpha\to\beta).\lambda x : \alpha.f(gx)$$

Type:

$$\forall\alpha.\forall\beta.\forall\gamma.(\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma$$

Usage: compose[Nat][Nat][Bool] isZero succ

Encodings in System F

Encoding: Church Booleans (1/3)

Recall Church Booleans in untyped λ-calculus:

- $\text{True} = \lambda t.\lambda f.t$
- $\text{False} = \lambda t.\lambda f.f$
- $\text{if} = \lambda b.\lambda t.\lambda e.b\ t\ e$ (or just apply b directly)

Why not in $\lambda_{\rightarrow}$?

- We can type True and False if t and f have the same type, e.g., $\text{T}\rightarrow\text{T}\rightarrow\text{T}$.
- But the *operators* (like `and`, `or`, `not`) are problematic.
- Example: $\texttt{and} = \lambda b_1.\lambda b_2.b_1\ b_2\ \text{False}$. From looking at the application of b_1: $\text{T} = \text{T}\rightarrow\text{T}\rightarrow\text{T}$, oops.

Encoding: Church Booleans (1/3)

Recall Church Booleans in untyped λ-calculus:

- $\mathsf{True} = \lambda t.\lambda f.t$
- $\mathsf{False} = \lambda t.\lambda f.f$
- $\mathsf{if} = \lambda b.\lambda t.\lambda e.b\ t\ e$ (or just apply b directly)

Why not in $\lambda_{\rightarrow}$?

- We can type True and False if t and f have the same type, e.g., $\mathsf{T}\rightarrow\mathsf{T}\rightarrow\mathsf{T}$.
- But the *operators* (like `and`, `or`, `not`) are problematic.
- Example: $\texttt{and} = \lambda b_1.\lambda b_2.b_1\ b_2\ \mathsf{False}$. From looking at the application of b_1: $\mathsf{T} = \mathsf{T}\rightarrow\mathsf{T}\rightarrow\mathsf{T}$, oops.

Intuition: A boolean, due to its encoding, must work with different types: e.g., used directly $b\ 0\ 1$, but also sometimes it is used to apply to another boolean, like in $b_1\ b_2\ \mathsf{false}$. In general, a boolean should take two arguments (then/else) of *any* type α and return one of these two arguments.

Encoding: Church Booleans (2/3)

Polymorphic Type for Booleans in System F:

$$\mathsf{Bool}_{\mathsf{Church}} = \forall\alpha.\alpha\to\alpha\to\alpha$$

Definitions:

$$\mathsf{True} = \Lambda\alpha.\lambda t : \alpha.\lambda f : \alpha.t$$

$$\mathsf{False} = \Lambda\alpha.\lambda t : \alpha.\lambda f : \alpha.f$$

Check type: $\emptyset \vdash \mathsf{True} : \mathsf{Bool}_{\mathsf{Church}}$ (similar derivation to identity)

Encoding: Church Booleans (3/3)

Now we *can* define operators:

$$\texttt{and} = \lambda b_1 : \mathsf{Bool}_{\mathsf{Church}}.\lambda b_2 : \mathsf{Bool}_{\mathsf{Church}}.b_1[\mathsf{Bool}_{\mathsf{Church}}]\ b_2\ \mathsf{False}$$

$$\texttt{and} : \mathsf{Bool}_{\mathsf{Church}} \rightarrow \mathsf{Bool}_{\mathsf{Church}} \rightarrow \mathsf{Bool}_{\mathsf{Church}}$$

Encoding: Church Booleans (3/3)

Now we *can* define operators:

$$\texttt{and} = \lambda b_1 : \mathsf{Bool}_{\mathsf{Church}}.\lambda b_2 : \mathsf{Bool}_{\mathsf{Church}}.b_1[\mathsf{Bool}_{\mathsf{Church}}]\ b_2\ \mathsf{False}$$

$$\texttt{and} : \mathsf{Bool}_{\mathsf{Church}} \to \mathsf{Bool}_{\mathsf{Church}} \to \mathsf{Bool}_{\mathsf{Church}}$$

Subtlety: The type application $b_1[\mathsf{Bool}_{\mathsf{Church}}]$ specializes the boolean b_1 to return... another boolean! This "self-reference" at the type level is key.

Encoding: Church Numerals (1/2)

Recall Church Numerals:

- $\bar{0} = \lambda s.\lambda z.z$
- $\bar{1} = \lambda s.\lambda z.sz$
- $\bar{n} = \lambda s.\lambda z.s^n z$

Intuition: A numeral $\bar{n}$ takes a successor function s and a zero value z, and applies s n times to z. The types of s and z should be flexible.

Polymorphic Type in System F:

$$\mathsf{Nat}_{\mathsf{Church}} = \forall\alpha.(\alpha\to\alpha)\to\alpha\to\alpha$$

Encoding: Church Numerals (1/2)

Recall Church Numerals:

- $\bar{0} = \lambda s.\lambda z.z$
- $\bar{1} = \lambda s.\lambda z.sz$
- $\bar{n} = \lambda s.\lambda z.s^n z$

Intuition: A numeral $\bar{n}$ takes a successor function s and a zero value z, and applies s n times to z. The types of s and z should be flexible.

Polymorphic Type in System F:

$$\mathsf{Nat}_{\mathsf{Church}} = \forall\alpha.(\alpha\to\alpha)\to\alpha\to\alpha$$

Definitions:

$$\bar{0} = \Lambda\alpha.\lambda s : (\alpha\to\alpha).\lambda z : \alpha.z$$

$$\mathsf{succ} = \lambda n : \mathsf{Nat}_{\mathsf{Church}}.\Lambda\alpha.\lambda s : (\alpha\to\alpha).\lambda z : \alpha.s(n[\alpha]\ s\ z)$$

Encoding: Church Numerals (2/2)

Definitions:

$$\bar{0} = \Lambda\alpha.\lambda s : (\alpha{\rightarrow}\alpha).\lambda z : \alpha.z$$

$$\mathsf{succ} = \lambda n : \mathsf{Nat}_{\mathsf{Church}}.\Lambda\alpha.\lambda s : (\alpha{\rightarrow}\alpha).\lambda z : \alpha.s(n[\alpha]\ s\ z)$$

Example: Plus

$$\mathsf{plus} = \lambda m : \mathsf{Nat}_{\mathsf{Church}}.\lambda n : \mathsf{Nat}_{\mathsf{Church}}.\\ \Lambda\alpha.\lambda s : (\alpha{\rightarrow}\alpha).\lambda z : \alpha.\\ m[\alpha]\ s\ (n[\alpha]\ s\ z)$$

Encoding: Pairs

Can we encode pairs without built-in product types ($\tau_1 \times \tau_2$)?

Encoding: Pairs

Can we encode pairs without built-in product types ($\tau_1 \times \tau_2$)?

Yes!

Encoding: Pairs

Can we encode pairs without built-in product types ($\tau_1 \times \tau_2$)?
Yes!

Intuition: A pair (a, b) is something that, when given a function f, applies f to a and b. The result type depends on f.

Encoding: Pairs

Can we encode pairs without built-in product types ($\tau_1 \times \tau_2$)?
Yes!

<u>Intuition:</u> A pair (a, b) is something that, when given a function f, applies f to a and b. The result type depends on f. Polymorphic Type:

$$\text{Pair}(\tau_1, \tau_2) = \forall\beta.(\tau_1 \rightarrow \tau_2 \rightarrow \beta) \rightarrow \beta$$

Encoding: Pairs

Can we encode pairs without built-in product types ($\tau_1 \times \tau_2$)?
Yes!

<u>Intuition:</u> A pair (a, b) is something that, when given a function f, applies f to a and b. The result type depends on f. Polymorphic Type:

$$\text{Pair}(\tau_1, \tau_2) = \forall\beta.(\tau_1 \to \tau_2 \to \beta) \to \beta$$

Constructor:

$$\text{mkpair} = \Lambda\tau_1.\Lambda\tau_2.\lambda x : \tau_1.\lambda y : \tau_2.\Lambda\beta.\lambda f : (\tau_1 \to \tau_2 \to \beta).f\ x\ y$$

$$\text{mkpair} : \forall\tau_1.\forall\tau_2.\tau_1 \to \tau_2 \to \text{Pair}(\tau_1, \tau_2)$$

Encoding: Pairs

Can we encode pairs without built-in product types ($\tau_1 \times \tau_2$)?
Yes!

<u>Intuition:</u> A pair (a, b) is something that, when given a function f, applies f to a and b. The result type depends on f. Polymorphic Type:

$$\mathsf{Pair}(\tau_1, \tau_2) = \forall \beta.(\tau_1 \to \tau_2 \to \beta) \to \beta$$

Constructor:

$$\mathsf{mkpair} = \Lambda\tau_1.\Lambda\tau_2.\lambda x : \tau_1.\lambda y : \tau_2.\Lambda\beta.\lambda f : (\tau_1 \to \tau_2 \to \beta).f\ x\ y$$

$$\mathsf{mkpair} : \forall \tau_1.\forall \tau_2.\tau_1 \to \tau_2 \to \mathsf{Pair}(\tau_1, \tau_2)$$

Projections:

$$\mathsf{fst} = \Lambda\tau_1.\Lambda\tau_2.\lambda p : \mathsf{Pair}(\tau_1, \tau_2).p[\tau_1]\ (\lambda x : \tau_1.\lambda y : \tau_2.x)$$

$$\mathsf{fst} : \forall \tau_1.\forall \tau_2.\mathsf{Pair}(\tau_1, \tau_2) \to \tau_1$$

Abstract Data Types via Existentials

Goal: Bundle a hidden representation type (ρ) with operations acting on it.

Informal Idea (Stack): We want a type that means: "There exists some type ρ (the stack representation), such that we have:

- An empty element of type ρ.
- A push operation: $\rho \to \mathsf{Int} \to \rho$.
- A pop operation: $\rho \to \mathsf{Option}(\mathsf{Int} \times \rho)$."

This is written conceptually as:

$\exists \rho.\{\mathsf{empty} : \rho, \mathsf{push} : \rho \to \mathsf{Int} \to \rho, \mathsf{pop} : \rho \to \mathsf{Option}(\mathsf{Int} \times \rho)\}$

Abstract Data Types via Existentials

Goal: Bundle a hidden representation type (ρ) with operations acting on it.

Informal Idea (Stack): We want a type that means: "There exists some type ρ (the stack representation), such that we have:

- An empty element of type ρ.
- A push operation: $\rho \to \mathsf{Int} \to \rho$.
- A pop operation: $\rho \to \mathsf{Option}(\mathsf{Int} \times \rho)$."

This is written conceptually as:

$\exists \rho.\{\mathsf{empty} : \rho, \mathsf{push} : \rho \to \mathsf{Int} \to \rho, \mathsf{pop} : \rho \to \mathsf{Option}(\mathsf{Int} \times \rho)\}$

We need to encode this $\exists$ using $\forall$ in System F.

Stack ADT - Encoding the Type

Let's define the interface signature type (dependent on ρ):

$$\mathsf{StackInterface}(\rho) = \{\mathsf{empty} : \rho, \mathsf{push} : \rho \to \mathsf{Int} \to \rho, \mathsf{pop} : \ldots\}$$

(Assuming a record type '...' exists or is encoded in System F)

Stack ADT - Encoding the Type

Let's define the interface signature type (dependent on ρ):

$$\text{StackInterface}(\rho) = \{\text{empty} : \rho, \text{push} : \rho \to \text{Int} \to \rho, \text{pop} : ...\}$$

(Assuming a record type '...' exists or is encoded in System F)

Now, encode the existential $\exists \rho.\text{StackInterface}(\rho)$ using the universal quantifier:

$$\text{StackADT} = \forall \alpha.(\forall \rho.(\text{StackInterface}(\rho) \to \alpha)) \to \alpha$$

Stack ADT - Encoding the Type

Let's define the interface signature type (dependent on ρ):

$$\text{StackInterface}(\rho) = \{\text{empty} : \rho, \text{push} : \rho \to \text{Int} \to \rho, \text{pop} : ...\}$$

(Assuming a record type '...' exists or is encoded in System F)

Now, encode the existential $\exists \rho.\text{StackInterface}(\rho)$ using the universal quantifier:

$$\text{StackADT} = \forall \alpha.(\forall \rho.(\text{StackInterface}(\rho) \to \alpha)) \to \alpha$$

Let's explain with an example this weird type:

- how to produce a StackADT.
- how to use a StackADT.

Stack ADT - Implementation

Let's choose a concrete representation: $\rho = \mathsf{List}(\mathsf{Int})$ (assume Lists are encoded).
First, implement the interface for $\mathsf{List}(\mathsf{Int})$:

Stack ADT - Implementation

Let's choose a concrete representation: $\rho = \text{List(Int)}$ (assume Lists are encoded).
First, implement the interface for List(Int):

```
concreteEmpty : List(Int) = Nil
concretePush : List(Int) -> Int -> List(Int) =
  fun s i -> Cons i s
concretePop : List(Int) -> Option(Int * List(Int)) =
  fun s -> case s of
             Nil      -> None
           | Cons h t ->  or Some (mkpair h t)

concreteIFace : StackInterface(List(Int)) =
  { empty = concreteEmpty,
    push = concretePush,
    pop = concretePop }
```

Stack ADT - Implementation Packing

Now, pack this implementation into the ‘StackADT‘ type:

$$\begin{aligned}
&\mathit{mkListStack} : \mathsf{StackADT} = \\
&\qquad \lambda\alpha.\lambda k : (\forall\rho.(\mathsf{StackInterface}(\rho) \to \alpha)). \\
&\qquad\qquad k[\mathsf{List}(\mathsf{Int})]\ \mathit{concreteIFace}
\end{aligned}$$

Stack ADT - Implementation Packing

Now, pack this implementation into the ‘StackADT‘ type:

$$\begin{aligned} &\mathit{mkListStack} : \mathsf{StackADT} = \\ &\qquad \lambda\alpha.\lambda \mathit{k} : (\forall\rho.(\mathsf{StackInterface}(\rho) \to \alpha)). \\ &\qquad\qquad \mathit{k}[\mathsf{List}(\mathsf{Int})]\ \mathit{concreteIFace} \end{aligned}$$

The value ‘mkListStack‘ has type ‘StackADT‘. Its user doesn’t know $\rho = \mathsf{List}(\mathsf{Int})$.

Stack ADT - Usage (Unpacking)

A client has a value, say ‘myStack : StackADT‘.

Stack ADT - Usage (Unpacking)

A client has a value, say ‘myStack : StackADT‘.

Example: Compute a simple value, e.g., returns the value popped after push 1.

$$
\begin{aligned}
&\textit{useStack}(s : \mathsf{StackADT}) = \\
&\qquad s[\mathsf{Option}(\mathsf{Int})] \\
&\qquad\quad (\Lambda\rho.\lambda \mathsf{iface} : \mathsf{StackInterface}(\rho). \\
&\qquad\qquad \mathsf{let}\ s_0 = \mathsf{iface.empty}\ \mathsf{in} \\
&\qquad\qquad \mathsf{let}\ s_1 = \mathsf{iface.push}\ s_0\ 1\ \mathsf{in} \\
&\qquad\qquad \mathsf{match}\ (\mathsf{iface.pop}\ s_1)\ \mathsf{with} \\
&\qquad\qquad\quad \mathsf{None} \to \mathsf{None} \\
&\qquad\qquad\quad |\mathsf{Some}\ p \to \mathsf{Some}\ (p.1)\ \mathsf{end})
\end{aligned}
$$

Stack ADT - Interpretation

$$\text{StackADT} = \forall\alpha.(\forall\rho.(\text{StackInterface}(\rho) \to \alpha)) \to \alpha$$

Interpretation:

- A value of type ‘StackADT‘ is a "package".
- To use the package (to get a result of type α), you must provide a function (the continuation k).
- This function k must be polymorphic in the hidden representation ρ ($\forall\rho$). It takes the interface for that ρ and produces an α.
- The package, when opened, applies the user's universal function k to its specific hidden representation type and its concrete interface implementation.

Stack ADT - Interpretation

$$\text{StackADT} = \forall \alpha.(\forall \rho.(\text{StackInterface}(\rho) \rightarrow \alpha)) \rightarrow \alpha$$

Interpretation:

- A value of type ‘StackADT‘ is a ”package”.
- To use the package (to get a result of type α), you must provide a function (the continuation k).
- This function k must be polymorphic in the hidden representation ρ ($\forall \rho$). It takes the interface for that ρ and produces an α.
- The package, when opened, applies the user's universal function k to its specific hidden representation type and its concrete interface implementation.
- Limitations?

Metatheory of System F

Metatheory: Soundness

Like $\lambda_{\rightarrow}$, System F enjoys safety properties:

- **Progress**: A well-typed term t (where t is not a value) can take a step: $t \longrightarrow t'$.
- **Preservation**: If $\Gamma \vdash t : \tau$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : \tau$.

Proof differences from $\lambda_{\rightarrow}$:

Metatheory: Soundness

Like $\lambda_{\rightarrow}$, System F enjoys safety properties:

- **Progress**: A well-typed term t (where t is not a value) can take a step: $t \longrightarrow t'$.
- **Preservation**: If $\Gamma \vdash t : \tau$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : \tau$.

Proof differences from $\lambda_{\rightarrow}$:

- Need to handle the new syntax: type abstractions ($\Lambda\alpha.t$) and type applications ($t[\tau]$).
- Need corresponding cases in the proofs (e.g., for T-TAbs, T-TApp rules in Preservation).

Metatheory: Soundness

Like $\lambda_{\rightarrow}$, System F enjoys safety properties:

- **Progress**: A well-typed term t (where t is not a value) can take a step: $t \longrightarrow t'$.
- **Preservation**: If $\Gamma \vdash t : \tau$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : \tau$.

Proof differences from $\lambda_{\rightarrow}$:

- Need to handle the new syntax: type abstractions ($\Lambda\alpha.t$) and type applications ($t[\tau]$).
- Need corresponding cases in the proofs (e.g., for T-TAbs, T-TApp rules in Preservation).
- Requires lemmas about substitution involving types (e.g., type substitution preserves typing).

Metatheory: Soundness

Like $\lambda_{\rightarrow}$, System F enjoys safety properties:

- **Progress**: A well-typed term t (where t is not a value) can take a step: $t \longrightarrow t'$.
- **Preservation**: If $\Gamma \vdash t : \tau$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : \tau$.

Proof differences from $\lambda_{\rightarrow}$:

- Need to handle the new syntax: type abstractions ($\Lambda\alpha.t$) and type applications ($t[\tau]$).
- Need corresponding cases in the proofs (e.g., for T-TAbs, T-TApp rules in Preservation).
- Requires lemmas about substitution involving types (e.g., type substitution preserves typing).
- Handling of environment can become quite technical, depending on the encoding. Maybe having two environment is easier.

Metatheory: Soundness

Like $\lambda_{\rightarrow}$, System F enjoys safety properties:

- **Progress**: A well-typed term t (where t is not a value) can take a step: $t \longrightarrow t'$.
- **Preservation**: If $\Gamma \vdash t : \tau$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : \tau$.

Proof differences from $\lambda_{\rightarrow}$:

- Need to handle the new syntax: type abstractions ($\Lambda\alpha.t$) and type applications ($t[\tau]$).
- Need corresponding cases in the proofs (e.g., for T-TAbs, T-TApp rules in Preservation).
- Requires lemmas about substitution involving types (e.g., type substitution preserves typing).
- Handling of environment can become quite technical, depending on the encoding. Maybe having two environment is easier.

Result: Well-typed System F programs do not get stuck.

Metatheory: Strong Normalization

Recall Strong Normalization (SN) for $\lambda_{\rightarrow}$: All well-typed terms terminate (evaluation reaches a value). Does System F have

Strong Normalization?

Metatheory: Strong Normalization

Recall Strong Normalization (SN) for $\lambda_{\rightarrow}$: All well-typed terms terminate (evaluation reaches a value). Does System F have

Strong Normalization? **Yes!** (Girard 1972)

Metatheory: Strong Normalization

Recall Strong Normalization (SN) for $\lambda_{\rightarrow}$: All well-typed terms terminate (evaluation reaches a value). Does System F have

Strong Normalization? **Yes!** (Girard 1972)

Attempting to extend the $\lambda_{\rightarrow}$ proof method naively fails.

- The logical relation in $\lambda_{\rightarrow}$ is defined inductively on the structure of types.
- How to define the relation for $\forall\alpha.\tau$?

Metatheory: Strong Normalization

Recall Strong Normalization (SN) for $\lambda_{\rightarrow}$: All well-typed terms terminate (evaluation reaches a value). Does System F have

Strong Normalization? **Yes!** (Girard 1972)

Attempting to extend the $\lambda_{\rightarrow}$ proof method naively fails.

- The logical relation in $\lambda_{\rightarrow}$ is defined inductively on the structure of types.
- How to define the relation for $\forall\alpha.\tau$?
- $R_{\forall\alpha.F} = \{u | \forall T, uT \in R_{F[\alpha\rightarrow T]}\}$?

Metatheory: Strong Normalization

Recall Strong Normalization (SN) for $\lambda_\rightarrow$: All well-typed terms terminate (evaluation reaches a value). Does System F have

Strong Normalization? **Yes!** (Girard 1972)

Attempting to extend the $\lambda_\rightarrow$ proof method naively fails.

- The logical relation in $\lambda_\rightarrow$ is defined inductively on the structure of types.
- How to define the relation for $\forall\alpha.\tau$?
- $R_{\forall\alpha.F} = \{u | \forall T, uT \in R_{F[\alpha\rightarrow T]}\}$? Fishy. It is a circular definition : $R_{\forall\alpha.\alpha}$ is defined from itself (take $T = \forall\alpha.\alpha$)!

Metatheory: Strong Normalization

Recall Strong Normalization (SN) for $\lambda_{\rightarrow}$: All well-typed terms terminate (evaluation reaches a value). Does System F have

Strong Normalization? **Yes!** (Girard 1972)

Attempting to extend the $\lambda_{\rightarrow}$ proof method naively fails.

- The logical relation in $\lambda_{\rightarrow}$ is defined inductively on the structure of types.
- How to define the relation for $\forall\alpha.\tau$?
- $R_{\forall\alpha.F} = \{u | \forall T, uT \in R_{F[\alpha\rightarrow T]}\}$? Fishy. It is a circular definition : $R_{\forall\alpha.\alpha}$ is defined from itself (take $T = \forall\alpha.\alpha$)!
- Girard's proof requires a trick/proof technique (reducibility candidates).

System F is significantly more powerful than $\lambda_{\rightarrow}$, but still guarantees termination.

Metatheory: Evaluation of System F

Naive evaluation:

- Follows the reduction rules directly (E-AppAbs, E-TAppTAbs).
- Carry types around at runtime.
- Perform substitutions in types and terms.

Metatheory: Evaluation of System F

Naive evaluation:

- Follows the reduction rules directly (E-AppAbs, E-TAppTAbs).
- Carry types around at runtime.
- Perform substitutions in types and terms.

Observation: Type information ($\Lambda\alpha$, $t[\tau]$) guides reduction but doesn't change "computational content".

- There is no reduction *within* types themselves in System F.
- Is carrying all this type information strictly necessary for computation?

Metatheory: Evaluation of System F with erasure

Introduce **Erasure**: A function $\text{erase}(t)$ that removes all type annotations and operations.

- $\text{erase}(\lambda x : \tau.t) = \lambda x.\text{erase}(t)$
- $\text{erase}(t_1\ t_2) = \text{erase}(t_1)\ \text{erase}(t_2)$
- $\text{erase}(\Lambda\alpha.t) = \text{erase}(t)$
- $\text{erase}(t[\tau]) = \text{erase}(t)$

Result: $\text{erase}(t)$ is an untyped lambda calculus term. Evaluation in System F simulates evaluation in untyped λ-calculus after erasure.

Metatheory: Type Inference / Reconstruction

Erasure maps a System F term to an untyped term.

Question: Is erasure always invertible? Given an untyped term u, can we find a System F term t such that $\text{erase}(t) = u$ and $\emptyset \vdash t : \tau$ for some τ? (Type Reconstruction/Inference)

Metatheory: Type Inference / Reconstruction

Erasure maps a System F term to an untyped term.
Question: Is erasure always invertible? Given an untyped term u, can we find a System F term t such that $\text{erase}(t) = u$ and $\emptyset \vdash t : \tau$ for some τ? (Type Reconstruction/Inference)

Answer: Clearly not! Consider the term Ω.

Metatheory: Type Inference / Reconstruction

Erasure maps a System F term to an untyped term.
Question: Is erasure always invertible? Given an untyped term u, can we find a System F term t such that $\text{erase}(t) = u$ and $\emptyset \vdash t : \tau$ for some τ? (Type Reconstruction/Inference)

Answer: Clearly not! Consider the term Ω. More interestingly, for a given term, deciding if there exists a preimage in System F is **Undecidable**.

Metatheory: Type Inference / Reconstruction

Erasure maps a System F term to an untyped term.
Question: Is erasure always invertible? Given an untyped term u, can we find a System F term t such that $\text{erase}(t) = u$ and $\emptyset \vdash t : \tau$ for some τ? (Type Reconstruction/Inference)

Answer: Clearly not! Consider the term Ω. More interestingly, for a given term, deciding if there exists a preimage in System F is **Undecidable**.

Why is it hard?

- Where to put $\Lambda\alpha$ and $t[\tau]$? Many possibilities.
- Determining the polymorphic types ($\forall$) is complex.
- Requires higher-order unification in general.

Metatheory: Type Inference / Reconstruction

Erasure maps a System F term to an untyped term.
Question: Is erasure always invertible? Given an untyped term u, can we find a System F term t such that $\text{erase}(t) = u$ and $\emptyset \vdash t : \tau$ for some τ? (Type Reconstruction/Inference)

Answer: Clearly not! Consider the term Ω. More interestingly, for a given term, deciding if there exists a preimage in System F is **Undecidable**.

Why is it hard?

- Where to put $\Lambda\alpha$ and $t[\tau]$? Many possibilities.
- Determining the polymorphic types ($\forall$) is complex.
- Requires higher-order unification in general.

This is why languages like Haskell and ML use restricted forms of polymorphism (like Hindley-Milner / Rank-1 polymorphism) where type inference is decidable. System F is too expressive for full inference.

Parametricity and Theorems for Free (1/2)

Consider the type $\forall\alpha.\alpha\to\alpha$. What terms have this type?

Parametricity and Theorems for Free (1/2)

Consider the type $\forall\alpha.\alpha\to\alpha$. What terms have this type?

Only the identity function ($\Lambda\alpha.\lambda x : \alpha.x$), module reduction. Why? Because the function must work uniformly for *all* types α. It cannot inspect the type α or behave differently based on it. It can only pass the value x through.

Parametricity and Theorems for Free (1/2)

Consider the type $\forall\alpha.\alpha\to\alpha$. What terms have this type?

Only the identity function ($\Lambda\alpha.\lambda x : \alpha.x$), module reduction. Why? Because the function must work uniformly for *all* types α. It cannot inspect the type α or behave differently based on it. It can only pass the value x through.

Similarly, consider $\forall\alpha.\alpha\to\alpha\to\alpha$. What are the possible terms?

Parametricity and Theorems for Free (1/2)

Consider the type $\forall\alpha.\alpha\to\alpha$. What terms have this type?

Only the identity function ($\Lambda\alpha.\lambda x:\alpha.x$), module reduction. Why? Because the function must work uniformly for *all* types α. It cannot inspect the type α or behave differently based on it. It can only pass the value x through.

Similarly, consider $\forall\alpha.\alpha\to\alpha\to\alpha$. What are the possible terms? $\Lambda\alpha.\lambda x:\alpha.\lambda y:\alpha.x$ and $\Lambda\alpha.\lambda x:\alpha.\lambda y:\alpha.y$.

Parametricity and Theorems for Free (2/2)

Consider $\texttt{map} : \forall\alpha.\forall\beta.(\alpha\to\beta)\to\mathsf{List}(\alpha)\to\mathsf{List}(\beta)$. Parametricity tells us properties this function *must* have, e.g.,

$$\texttt{map}(f \circ g) = \texttt{map}(f) \circ \texttt{map}(g)$$

Parametricity and Theorems for Free (2/2)

Consider $\texttt{map} : \forall\alpha.\forall\beta.(\alpha\to\beta)\to\mathsf{List}(\alpha)\to\mathsf{List}(\beta)$. Parametricity tells us properties this function *must* have, e.g.,

$$\texttt{map}(f \circ g) = \texttt{map}(f) \circ \texttt{map}(g)$$

These properties arise "for free" just from the polymorphic type, without looking at the implementation.

Parametricity and Theorems for Free (2/2)

Consider $\mathtt{map} : \forall\alpha.\forall\beta.(\alpha\to\beta)\to\mathsf{List}(\alpha)\to\mathsf{List}(\beta)$. Parametricity tells us properties this function *must* have, e.g.,

$$\mathtt{map}(f \circ g) = \mathtt{map}(f) \circ \mathtt{map}(g)$$

These properties arise "for free" just from the polymorphic type, without looking at the implementation.

Intuition: Universal quantification ($\forall\alpha$) provides strong guarantees. A function polymorphic in α must treat values of type α abstractly, leading to uniform behavior across all types. *This gives semantic guarantees beyond just type safety.*

Curry-Howard for System F

Recall Curry-Howard for $\lambda_{\rightarrow}$:

- Types $\leftrightarrow$ Propositions
- Terms $\leftrightarrow$ Proofs
- $\tau_1 \rightarrow \tau_2 \leftrightarrow P_1 \Rightarrow P_2$ (Implication)
- $\tau_1 \times \tau_2 \leftrightarrow P_1 \wedge P_2$ (Conjunction)
- $\tau_1 + \tau_2 \leftrightarrow P_1 \vee P_2$ (Disjunction)

Well-typed terms correspond to constructive proofs in intuitionistic propositional logic.

What about the new rules in System F?

- Type variable $\alpha \leftrightarrow$ Propositional variable A
- Type abstraction $\Lambda\alpha.t \leftrightarrow$ Universal quantification introduction ($\forall$)
- Type application $t[\tau] \leftrightarrow$ Universal quantification elimination ($\forall$-elim)

So, $\forall\alpha.\tau \leftrightarrow \forall A.P$

System F corresponds to **Second-Order Intuitionistic Propositional Logic**.

Beyond System F

System F (parametric polymorphism) is powerful, but other forms exist:

- **Overloading:** Functions with the same name behave differently based on static argument types (e.g., '+' for Ints and Floats). Often handled via mechanisms like type classes (Haskell) or implicit parameters (Scala).
- **Subtype Polymorphism:** If τ_1 is a subtype of τ_2 ($\tau_1 <: \tau_2$), then a value of type τ_1 can be used where a value of type τ_2 is expected. Common in object-oriented languages.

These can be combined, leading to systems like System $F_{<:}$ (System F with subtyping).