Programming Language Theory

Higher-Order Polymorphism
Looking back, looking forward

Have defined System F.

- Metatheory (what properties does it have)
- What (else) is it good for
- How/why ML is more restrictive and implicit
- Recursive types (also use type variables, but differently)
- Existential types (dual to universal types)

Next:

- Type operators and type-level “computations”
System F with Recursive and Existential Types

\[
e ::= \ c | x | \lambda x: \tau. \ e | e \ e | \\
\Lambda \alpha. \ e | e \ [\tau] | \\
\text{pack}_\exists \alpha. \ \tau(\tau, e) | \text{unpack} \ e \ \text{as} \ (\alpha, x) \ \text{in} \ e \\
\text{roll}_{\mu \alpha}. \ \tau(e) | \text{unroll}(e)
\]

\[
v ::= \ c | \lambda x: \tau. \ e | \Lambda \alpha. \ e | \text{pack}_\exists \alpha. \ \tau(\tau, v) | \text{roll}_{\mu \alpha}. \ \tau(v)
\]

\[
e \rightarrow_{\text{cbv}} e' \\
\frac{e_f \rightarrow_{\text{cbv}} e'_f}{e_f \ e_a \rightarrow_{\text{cbv}} e'_f \ e_a} \\
\frac{e_a \rightarrow_{\text{cbv}} e'_a}{\nu_f \ e_a \rightarrow_{\text{cbv}} \nu_f \ e'_a} \\
\frac{(\lambda x: \tau. \ e_b) \ v_a \rightarrow_{\text{cbv}} e_b[v_a/x]}{(\lambda x: \tau. \ e_b) \ v_a \rightarrow_{\text{cbv}} e_b[v_a/x]} \\
\frac{e_f \rightarrow_{\text{cbv}} e'_f}{e_f \ [\tau_a] \rightarrow_{\text{cbv}} e'_f \ [\tau_a]} \\
\frac{(\Lambda \alpha. \ e_b) \ [\tau_a] \rightarrow_{\text{cbv}} e_b[\tau_a/\alpha]}{(\Lambda \alpha. \ e_b) \ [\tau_a] \rightarrow_{\text{cbv}} e_b[\tau_a/\alpha]} \\
\frac{e_a \rightarrow_{\text{cbv}} e'_a}{\text{pack}_\exists \alpha. \ \tau(\tau_w, e_a) \rightarrow_{\text{cbv}} \text{pack}_\exists \alpha. \ \tau(\tau_w, e'_a)} \\
\frac{\text{unpack} \ e_a \ \text{as} \ (\alpha, x) \ \text{in} \ e_b \rightarrow_{\text{cbv}} \text{unpack} \ e'_a \ \text{as} \ (\alpha, x) \ \text{in} \ e_b}{\text{unpack} \ e_a \ \text{as} \ (\alpha, x) \ \text{in} \ e_b \rightarrow_{\text{cbv}} e_b[\tau_w/\alpha][v_a/x]} \\
\frac{e_a \rightarrow_{\text{cbv}} e'_a}{\text{roll}_{\mu \alpha}. \ \tau(e_a) \rightarrow_{\text{cbv}} \text{roll}_{\mu \alpha}. \ \tau(e'_a)} \\
\frac{e_a \rightarrow_{\text{cbv}} e'_a}{\text{unroll}(e_a) \rightarrow_{\text{cbv}} \text{unroll}(e'_a)} \\
\frac{\text{unroll}(\text{roll}_{\mu \alpha}. \ \tau(v_a)) \rightarrow_{\text{cbv}} v_a}{\text{unroll}(\text{roll}_{\mu \alpha}. \ \tau(v_a)) \rightarrow_{\text{cbv}} v_a}
\]
System F with Recursive and Existential Types

\[ \tau ::= \text{int} | \tau \rightarrow \tau | \alpha | \forall \alpha. \tau | \exists \alpha. \tau | \mu \alpha. \tau \]

\[ \Delta ::= \cdot | \Delta, \alpha \]

\[ \Gamma ::= \cdot | \Gamma, x: \tau \]

**Inference Rules**

\[ \Delta; \Gamma \vdash e : \tau \]

\[ \Delta; \Gamma \vdash \tau \]

\[ \Delta; \Gamma \vdash c : \text{int} \]

\[ \Delta; \Gamma \vdash \lambda x: \tau_a. \ e_b : \tau_r \]

\[ \Delta; \Gamma \vdash \text{pack}_{\exists \alpha. \tau} (\tau_w, e_a) : \exists \alpha. \tau \]

\[ \Delta; \Gamma \vdash \text{unroll} (e_a) : \tau \]

\[ \Delta; \Gamma \vdash \exists \alpha. \tau \]

\[ \Delta, x : \tau \vdash e_b : \tau_r \]

\[ \Delta, \alpha ; \Gamma \vdash e_b : \tau_r \]

\[ \Delta, \alpha ; \Gamma \vdash e_b : \tau_r \]

\[ \Delta ; \Gamma \vdash e_f : \tau_a \rightarrow \tau_r \]

\[ \Delta ; \Gamma \vdash e_f e_a : \tau_r \]

\[ \Delta ; \Gamma \vdash \text{unroll} (e_a) : \tau \]

\[ \Delta ; \Gamma \vdash r_{\mu \alpha. \tau} (e_a) : \mu \alpha. \tau \]

\[ \Delta, \alpha ; \Gamma \vdash e_a : \mu \alpha. \tau \]

\[ \Delta, \alpha ; \Gamma \vdash e_a : \mu \alpha. \tau \]

\[ \Delta, \alpha ; \Gamma \vdash e_a : \mu \alpha. \tau \]

\[ \Delta, \alpha ; \Gamma \vdash e_a : \mu \alpha. \tau \]

\[ \Delta, \alpha ; \Gamma \vdash e_a : \mu \alpha. \tau \]

\[ \Delta, \alpha ; \Gamma \vdash e_a : \mu \alpha. \tau \]

\[ \Delta, \alpha ; \Gamma \vdash e_a : \mu \alpha. \tau \]

\[ \Delta, \alpha ; \Gamma \vdash e_a : \mu \alpha. \tau \]

\[ \Delta, \alpha ; \Gamma \vdash e_a : \mu \alpha. \tau \]

\[ \Delta, \alpha ; \Gamma \vdash e_a : \mu \alpha. \tau \]

\[ \Delta, \alpha ; \Gamma \vdash e_a : \mu \alpha. \tau \]

\[ \Delta, \alpha ; \Gamma \vdash e_a : \mu \alpha. \tau \]

\[ \Delta, \alpha ; \Gamma \vdash e_a : \mu \alpha. \tau \]

\[ \Delta, \alpha ; \Gamma \vdash e_a : \mu \alpha. \tau \]

\[ \Delta, \alpha ; \Gamma \vdash e_a : \mu \alpha. \tau \]
Goal

Understand what this interface means and why it matters:

```ocaml
type 'a list
val empty : 'a list
val cons : 'a -> 'a list -> 'a list
val unlist : 'a list -> ('a * 'a list) option
val size : 'a list -> int
val map : ('a -> 'b) -> 'a list -> 'b list
```

Story so far:

- Recursive types to define list data structure
- Universal types to keep element type abstract in library
- Existential types to keep list type abstract in client

But, “cheated” when abstracting the list type in client: considered just intlist.
(Integer) List Library with ∃

List library is an existential package:

\[
\text{pack}(\mu \xi. \text{unit} + (\text{int} \times \xi), \text{list\_library}) \\
\text{as } \exists L. \{ \text{empty} : L; \\
\text{cons} : \text{int} \to L \to L; \\
\text{unlist} : L \to \text{unit} + (\text{int} \times L); \\
\text{map} : (\text{int} \to \text{int}) \to L \to L; \\
\ldots \}
\]

The witness type is integer lists: \( \mu \xi. \text{unit} + (\text{int} \times \xi) \).

The existential type variable \( L \) represents integer lists.

List operations are monomorphic in element type (\text{int}).

The \text{map} function only allows mapping integer lists to integer lists.
(Polymorphic?) List Library with ∀/∃

List library is a type abstraction that yields an existential package:

\[ \Lambda \alpha. \text{pack}(\mu \xi. \text{unit} + (\alpha \times \xi), \text{list\_library}) \]

as \( \exists L. \{ \text{empty} : L; \text{cons} : \alpha \to L \to L; \text{unlist} : L \to \text{unit} + (\alpha \times L); \text{map} : (\alpha \to \alpha) \to L \to L; \ldots \}\)

The witness type is \( \alpha \) lists: \( \mu \xi. \text{unit} + (\alpha \times \xi) \).

The existential type variable \( L \) represents \( \alpha \) lists.

List operations are monomorphic in element type \((\alpha)\).

The \textit{map} function only allows mapping \( \alpha \) lists to \( \alpha \) lists.
Type Abbreviations and Type Operators

Reasonable enough to provide list type as a \textit{(parametric) type abbreviation}:

\[
L\,\alpha = \mu\xi. \text{unit} + (\alpha \ast \xi)
\]

- replace occurrences of $L\,\tau$ in programs with $(\mu\xi. \text{unit} + (\alpha \ast \xi))[\tau/\alpha]$

Gives an \textit{informal} notion of functions at the type-level.

But, doesn’t help with list library, because this exposes the definition of list type.

- How “modular” and “safe” are libraries built from \texttt{cpp} macros?
Type Abbreviations and Type Operators

Instead, provide list type as a *type operator*:

- a function from types to types

\[
\text{\(L = \lambda \alpha. \mu \xi. \text{unit} + (\alpha \ast \xi)\)}
\]

Gives a *formal* notion of functions at the type-level.

- abstraction and application at the type-level
- equivalence of type-level expressions
- well-formedness of type-level expressions

List library will be an existential package that hides a *type operator*, (rather than a *type*).
Type-level Expressions

Abstraction and application at the type level makes it possible to write the same type with different syntax.

\[
\text{Id} = \lambda \alpha. \alpha
\]

\[
\begin{align*}
\text{int} \to \text{bool} & \quad \text{int} \to \text{Id bool} & \quad \text{Id int} \to \text{bool} & \quad \text{Id int} \to \text{Id bool} \\
\text{Id (int} \to \text{bool}) & \quad \text{Id (Id (int} \to \text{bool)}) & \quad \ldots
\end{align*}
\]
Type-level Expressions

Abstraction and application at the type level makes it possible to write the same type with different syntax.

\[
\text{Id} = \lambda \alpha. \alpha
\]

\[
\begin{align*}
\text{int} \rightarrow \text{bool} & \quad \text{int} \rightarrow \text{Id bool} & \quad \text{Id int} \rightarrow \text{bool} & \quad \text{Id int} \rightarrow \text{Id bool} \\
\text{Id (int} \rightarrow \text{bool}) & \quad \text{Id (Id (int} \rightarrow \text{bool})) & \quad \ldots
\end{align*}
\]

Require a precise definition of when two types are the same:

\[
\tau \equiv \tau'
\]

\[
\ldots \quad (\lambda \alpha. \tau_b) \tau_a \equiv \tau_b[\tau_a/\alpha] \quad \ldots
\]
Type-level Expressions

Abstraction and application at the type level makes it possible to write the same type with different syntax.

\[ \text{Id} = \lambda \alpha. \alpha \]

\[
\begin{align*}
\text{int} \to \text{bool} & \quad \text{int} \to \text{Id bool} & \quad \text{Id int} \to \text{bool} & \quad \text{Id int} \to \text{Id bool} \\
\text{Id (int} \to \text{bool)} & \quad \text{Id (Id (int} \to \text{bool)}) & \quad \ldots 
\end{align*}
\]

Require a typing rule to exploit types that are the same:

\[ \Delta; \Gamma \vdash e : \tau \]

\[ \begin{array}{c}
\Delta; \Gamma \vdash e : \tau \\
\tau \equiv \tau' \\
\end{array} \quad \Delta; \Gamma \vdash e : \tau' \quad \ldots \]
Type-level Expressions

Abstraction and application at the type level makes it possible to write the *same* type with *different* syntax.

\[ \text{Id} = \lambda \alpha. \alpha \]

\[
\begin{align*}
\text{int} \rightarrow \text{bool} & \quad \text{int} \rightarrow \text{Id bool} & \quad \text{Id int} \rightarrow \text{bool} & \quad \text{Id int} \rightarrow \text{Id bool} \\
\text{Id (int} \rightarrow \text{bool}) & \quad \text{Id (Id (int} \rightarrow \text{bool})) \quad \ldots 
\end{align*}
\]

Admits “wrong/bad/meaningless” types:

\[
\ldots \quad \text{bool int} \quad (\text{Id bool}) \text{ int} \quad \text{bool (Id int)} \quad \ldots
\]
Type-level Expressions

Abstraction and application at the type level makes it possible to write the same type with different syntax.

\[ \text{Id} = \lambda \alpha. \alpha \]

\[ \text{int} \to \text{bool} \quad \text{int} \to \text{Id bool} \quad \text{Id int} \to \text{bool} \quad \text{Id int} \to \text{Id bool} \]

\[ \text{Id (int \to \text{bool})} \quad \text{Id (Id (int \to \text{bool}))} \quad \ldots \]

Require a “type system” for types:

\[ \Delta \vdash \tau :: \kappa \]

\[ \ldots \]

\[ \Delta \vdash \tau_f :: \kappa_a \Rightarrow \kappa_r \quad \Delta \vdash \tau_a :: \kappa_a \]

\[ \Delta \vdash \tau_f \tau_a :: \kappa_r \]

\[ \ldots \]
Terms, Types, and Kinds, Oh My
Terms, Types, and Kinds, Oh My

Terms:  
- $e ::= c \mid x \mid \lambda x:\tau. \ e \mid e \ e \mid \Lambda \alpha::\kappa. \ e \mid e [\tau]$
- $v ::= c \mid \lambda x:\tau. \ e \mid \Lambda \alpha::\kappa. \ e$

- atomic values (e.g., $c$) and operations (e.g., $e + e$)
- compound values (e.g., $(v, v)$) and operations (e.g., $e.1$)
- value abstraction and application
- type abstraction and application
- classified by types (but not all terms have a type)
Terms, Types, and Kinds, Oh My

Terms:  
\[ e ::= c \mid x \mid \lambda x: \tau. \ e \mid e \ e \mid \Lambda \alpha::\kappa. \ e \mid e [\tau] \]
\[ \nu ::= c \mid \lambda x: \tau. \ e \mid \Lambda \alpha::\kappa. \ e \]

- atomic values (e.g., \( c \)) and operations (e.g., \( e + e \))
- compound values (e.g., \( (\nu, \nu) \)) and operations (e.g., \( e.1 \))
- value abstraction and application
- type abstraction and application
- classified by types (but not all terms have a type)

Types:  
\[ \tau ::= \text{int} \mid \tau \rightarrow \tau \mid \alpha \mid \forall \alpha::\kappa. \ \tau \mid \lambda \alpha::\kappa. \ \tau \mid \tau \ \tau \]

- atomic types (e.g., \( \text{int} \)) classify the terms that evaluate to atomic values
- compound types (e.g., \( \tau * \tau \)) classify the terms that evaluate to compound values
- function types \( \tau \rightarrow \tau \) classify the terms that evaluate to value abstractions
- universal types \( \forall \alpha. \ \tau \) classify the terms that evaluate to type abstractions
- type abstraction and application
  - type abstractions do not classify terms, but can be applied to type arguments to form types that do classify terms
- classified by kinds (but not all types have a kind)
Terms, Types, and Kinds, Oh My

Types: \( \tau ::= \text{int} | \tau \to \tau | \alpha | \forall \alpha :: \kappa. \tau | \lambda \alpha :: \kappa. \tau | \tau \tau \)

- atomic types (e.g., \text{int}) classify the terms that evaluate to atomic values
- compound types (e.g., \( \tau \star \tau \)) classify the terms that evaluate to compound values
- function types \( \tau \to \tau \) classify the terms that evaluate to value abstractions
- universal types \( \forall \alpha. \tau \) classify the terms that evaluate to type abstractions
- type abstraction and application
  - type abstractions do not classify terms, but can be applied to type arguments to form types that do classify terms
- classified by kinds (but not all types have a kind)
Terms, Types, and Kinds, Oh My

Types: \( \tau ::= \text{int} | \tau \rightarrow \tau | \alpha | \forall \alpha::\kappa. \tau | \lambda \alpha::\kappa. \tau | \tau \tau \)

- atomic types (e.g., int) classify the terms that evaluate to atomic values
- compound types (e.g., \( \tau * \tau \)) classify the terms that evaluate to compound values
- function types \( \tau \rightarrow \tau \) classify the terms that evaluate to value abstractions
- universal types \( \forall \alpha. \tau \) classify the terms that evaluate to type abstractions
- type abstraction and application
  - type abstractions do not classify terms, but can be applied to type arguments to form types that do classify terms
- classified by kinds (but not all types have a kind)

Kinds \( \kappa ::= \star | \kappa \Rightarrow \kappa \)

- kind of proper types \( \star \) classify the types (that are the same as the types) that classify terms
- arrow kinds \( \kappa \Rightarrow \kappa \) classify the types (that are the same as the types) that are type abstractions
Kind Examples

- the kind of proper types $\texttt{Bool}$, $\texttt{Bool} \to \texttt{Bool}$, ...

- the kind of (unary) type operators $\texttt{List}$, $\texttt{Maybe}$, ...

- the kind of (binary) type operators $\texttt{Either}$, $\texttt{Map}$, ...

- $(\star \to \star) \to \star$

- the kind of higher-order type operators taking unary type operators to proper types $\texttt{???,}$ ...

- $(\star \to \star) \to \star \to \star$

- the kind of higher-order type operators taking unary type operators to unary type operators $\texttt{MaybeT}$, $\texttt{ListT}$, ...
Kind Examples

- ★
  - the kind of proper types
  - \texttt{Bool}, \texttt{Bool} → \texttt{Bool}, . . .
Kind Examples

- ⭐
  - the kind of proper types
    - Bool, Bool → Bool, ...

- ⭐ ⇒ ⭐
  - the kind of (unary) type operators
    - List, Maybe, …
Kind Examples

- ★
  - the kind of proper types
    - `Bool, Bool → Bool, Maybe Bool, Maybe Bool → Maybe Bool, ...`
- ★ ⊢ ★
  - the kind of (unary) type operators
    - `List, Maybe, ...`
Kind Examples

- **the kind of proper types**
  - `Bool, Bool → Bool, Maybe Bool, Maybe Bool → Maybe Bool, ...`

- **the kind of (unary) type operators**
  - `List, Maybe, ...`

- **the kind of (binary) type operators**
  - `Either, Map, ...`
Kind Examples

- ★
  - the kind of proper types
  - `Bool, Bool → Bool, Maybe Bool, Maybe Bool → Maybe Bool, ...`

- ★ ⇒ ★
  - the kind of (unary) type operators
  - `List, Maybe, Map Int, Either (List Bool), ...`

- ★ ⇒ ★ ⇒ ★
  - the kind of (binary) type operators
  - `Either, Map, ...`
Kind Examples

- ★
  - the kind of proper types
  - `Bool, Bool → Bool, Maybe Bool, Maybe Bool → Maybe Bool, ...`

- ★ ⇒ ★
  - the kind of (unary) type operators
  - `List, Maybe, Map Int, Either (List Bool), ...`

- ★ ⇒ ★ ⇒ ★
  - the kind of (binary) type operators
  - `Either, Map, ...`

- (★ ⇒ ★) ⇒ ★
  - the kind of higher-order type operators
    - taking unary type operators to proper types
  - `???, ...`
Kind Examples

- ★
  - the kind of proper types
  - \(\text{Bool, Bool} \rightarrow \text{Bool, Maybe Bool, Maybe Bool} \rightarrow \text{Maybe Bool, \ldots}\)

- ★ ⇒ ★
  - the kind of (unary) type operators
  - \(\text{List, Maybe, Map Int, Either (List Bool), \ldots}\)

- ★ ⇒ ★ ⇒ ★
  - the kind of (binary) type operators
  - \(\text{Either, Map, \ldots}\)

- \((★ ⇒ ★) ⇒ ★\)
  - the kind of higher-order type operators
    - taking unary type operators to proper types
  - ???, \ldots

- \((★ ⇒ ★) ⇒ ★ ⇒ ★\)
  - the kind of higher-order type operators
    - taking unary type operators to unary type operators
  - \(\text{MaybeT, ListT, \ldots}\)
Kind Examples

- \(*\)
  - the kind of proper types
  - \(\text{Bool, Bool} \rightarrow \text{Bool, Maybe Bool, Maybe Bool} \rightarrow \text{Maybe Bool, \ldots}\)

- \(* \Rightarrow *\)
  - the kind of (unary) type operators
  - \(\text{List, Maybe, Map Int, Either (List Bool), ListT Maybe, \ldots}\)

- \(* \Rightarrow * \Rightarrow *\)
  - the kind of (binary) type operators
  - \(\text{Either, Map, \ldots}\)

- \((* \Rightarrow *) \Rightarrow *\)
  - the kind of higher-order type operators
taking unary type operators to proper types
  - \(???, \ldots\)

- \((* \Rightarrow *) \Rightarrow * \Rightarrow *\)
  - the kind of higher-order type operators
taking unary type operators to unary type operators
  - \(\text{MaybeT, ListT, \ldots}\)
System $F_\omega$: Syntax

\[
e ::= \ c \mid x \mid \lambda x: \tau. \ e \mid e \ e \mid \Lambda \alpha::\kappa. \ e \mid e [\tau]
\]

\[
v ::= \ c \mid \lambda x: \tau. \ e \mid \Lambda \alpha::\kappa. \ e
\]

\[
\Gamma ::= \cdot \mid \Gamma, x: \tau
\]

\[
\tau ::= \text{int} \mid \tau \rightarrow \tau \mid \alpha \mid \forall \alpha::\kappa. \ \tau \mid \lambda \alpha::\kappa. \ \tau \mid \tau \ \tau
\]

\[
\Delta ::= \cdot \mid \Delta, \alpha::\kappa
\]

\[
\kappa ::= \star \mid \kappa \Rightarrow \kappa
\]

New things:

- Types: type abstraction and type application
- Kinds: the “types” of types
  - $\star$: kind of proper types
  - $\kappa_a \Rightarrow \kappa_r$: kind of type operators
System $F_\omega$: Operational Semantics

Small-step, call-by-value (CBV), left-to-right operational semantics:

$e \rightarrow_{cbv} e'$

- $e_f \rightarrow_{cbv} e_f' \quad e_f \ e_a \rightarrow_{cbv} e'_f \ e'_a$
- $e_a \rightarrow_{cbv} e'_a \quad v_f \ e_a \rightarrow_{cbv} v_f \ e'_a$
- $(\lambda x : \tau. \ e_b) \ v_a \rightarrow_{cbv} e_b[v_a/x]$
- $v_f \ e_a \rightarrow_{cbv} v_f \ e'_a$
- $(\lambda x : \tau. \ e_b)[\tau_a] \rightarrow_{cbv} e_b[\tau_a/x]$
- $(\Lambda \alpha :: \kappa_a. \ e_b)[\tau_a] \rightarrow_{cbv} e_b[\tau_a/\alpha]$

▷ Unchanged! All of the new action is at the type-level.
System F_ω: Type System, part 1

In the context \( \Delta \) the type \( \tau \) has kind \( \kappa \):

\[
\Delta \vdash \tau :: \kappa
\]

\[
\begin{align*}
\Delta \vdash \text{int} :: \star \\
\Delta(\alpha) = \kappa \\
\Delta \vdash \alpha :: \kappa \\
\Delta, \alpha :: \kappa_a \vdash \tau_b :: \kappa_r \\
\Delta \vdash \lambda \alpha :: \kappa_a. \tau_b :: \kappa_a \Rightarrow \kappa_r \\
\end{align*}
\]

\[
\begin{align*}
\Delta \vdash \tau_a :: \star & \quad \Delta \vdash \tau_r :: \star \\
\Delta \vdash \tau_a \rightarrow \tau_r :: \star \\
\Delta, \alpha :: \kappa_a \vdash \tau_r :: \star \\
\Delta \vdash \forall \alpha :: \kappa_a. \tau_r :: \star \\
\Delta \vdash \tau_f :: \kappa_a \Rightarrow \kappa_r \\
\Delta \vdash \tau_a :: \kappa_a \\
\Delta \vdash \tau_f \tau_a :: \kappa_r \\
\end{align*}
\]

Should look familiar:
System F_ω: Type System, part 1

In the context $\Delta$ the type $\tau$ has kind $\kappa$:

$\Delta \vdash \tau :: \kappa$

\[
\begin{align*}
\Delta \vdash \text{int} :: \star & \quad \Delta \vdash \tau_a :: \star \quad \Delta \vdash \tau_r :: \star \\
\Delta \vdash \tau_a \rightarrow \tau_r :: \star & \\
\Delta \vdash \alpha :: \kappa & \\
\Delta, \alpha :: \kappa_a \vdash \tau_r :: \star & \\
\Delta \vdash \forall \alpha :: \kappa_a. \tau_r :: \star & \\
\Delta \vdash \tau_f :: \kappa_a \Rightarrow \kappa_r & \\
\Delta \vdash \tau_f \tau_a :: \kappa_r &
\end{align*}
\]

Should look familiar:
the typing rules of the Simply-Typed Lambda Calculus “one level up”
System $F_\omega$: Type System, part 2

*Definitional Equivalence of $\tau$ and $\tau'$:*

$\tau \equiv \tau'$

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>$\tau_1 \equiv \tau_2$</th>
<th>$\tau_2 \equiv \tau_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1 \equiv \tau_3$</td>
<td>$\tau_2 \equiv \tau_3$</td>
<td>$\tau_1 \equiv \tau_3$</td>
</tr>
</tbody>
</table>

$\tau_{a1} \equiv \tau_{a2} \quad \tau_{r1} \equiv \tau_{r2} \quad \forall \alpha :: \kappa_a . \tau_{r1} \equiv \forall \alpha :: \kappa_a . \tau_{r2}$

$\tau_{a1} \rightarrow \tau_{r1} \equiv \tau_{a2} \rightarrow \tau_{r2}$

$\lambda \alpha :: \kappa_a . \tau_{b1} \equiv \lambda \alpha :: \kappa_a . \tau_{b2}$

$(\lambda \alpha :: \kappa_a . \tau_b) \tau_a \equiv \tau_b[\tau_a / \alpha]$
System $F_\omega$: Type System, part 2

**Definitional Equivalence of $\tau$ and $\tau'$:**

$\boxed{\tau \equiv \tau'}$

\[
\begin{align*}
\tau &\equiv \tau \\
\tau_1 &\equiv \tau_2 \\
\tau_2 &\equiv \tau_1 \\
\tau_1 &\equiv \tau_3 \\
\tau_2 &\equiv \tau_3 \\
\forall \alpha::\kappa_a . \tau_r_1 &\equiv \forall \alpha::\kappa_a . \tau_r_2 \\
\tau_a_1 &\equiv \tau_a_2 \\
\tau_{r1} &\equiv \tau_{r2} \\
\tau_{a1} \rightarrow \tau_{r1} &\equiv \tau_{a2} \rightarrow \tau_{r2} \\
\tau_{b1} &\equiv \tau_{b2} \\
\lambda \alpha::\kappa_a . \tau_{b1} &\equiv \lambda \alpha::\kappa_a . \tau_{b2} \\
\tau_{f1} &\equiv \tau_{f2} \\
\tau_{a1} &\equiv \tau_{a2} \\
\tau_{f1} \tau_{a1} &\equiv \tau_{f2} \tau_{a2} \\
(\lambda \alpha::\kappa_a . \tau_b) \tau_a &\equiv \tau_b[\tau_a / \alpha]
\end{align*}
\]

Should look familiar:
the full reduction rules of the Lambda Calculus “one level up”
System $F_\omega$: Type System, part 3

In the contexts $\Delta$ and $\Gamma$ the expression $e$ has type $\tau$:

$$\Gamma(x) = \tau$$

$$\Delta; \Gamma \vdash x : \tau$$

$$\Delta; \Gamma, x : \tau_a \vdash e_b : \tau_r$$

$$\Delta; \Gamma \vdash \lambda x : \tau_a. \ e_b : \tau_a \to \tau_r$$

$$\Delta; \Gamma \vdash \tau_a :: \star$$

$$\Delta, \alpha :: \kappa_a; \Gamma \vdash e_b : \tau_r$$

$$\Delta; \Gamma \vdash \land \alpha. \ e_b : \forall \alpha :: \kappa_a. \ \tau_r$$

$$\Delta, \alpha :: \kappa_a; \Gamma \vdash e_b : \tau_r$$

$$\Delta; \Gamma \vdash e_f : \forall \alpha :: \kappa_a. \ \tau_r$$

$$\Delta; \Gamma \vdash e_f \ [\tau_a] : \tau_r[\tau_a/\alpha]$$

$$\Delta; \Gamma \vdash e : \tau \quad \tau \equiv \tau' \quad \Delta \vdash \tau' :: \star$$

$$\Delta; \Gamma \vdash e : \tau'$$

Syntax and type system easily extended with recursive and existential types.
System $F_\omega$: Type System, part 3

In the contexts $\Delta$ and $\Gamma$ the expression $e$ has type $\tau$:

$$\Delta; \Gamma \vdash e : \tau$$

$$\Gamma(x) = \tau \quad \Delta; \Gamma \vdash x : \tau$$

$$\Delta \vdash \tau_a :: \star \quad \Delta; \Gamma, x : \tau_a \vdash e_b : \tau_r \quad \Delta; \Gamma \vdash \lambda x : \tau_a. \ e_b : \tau_a \rightarrow \tau_r$$

$$\Delta, \alpha :: \kappa_a; \Gamma \vdash e_b : \tau_r \quad \Delta; \Gamma \vdash \forall \alpha :: \kappa_a. \ \tau \rightarrow \tau_r$$

$$\Delta; \Gamma \vdash \forall \alpha :: \kappa_a. \ \tau \quad \Delta \vdash \tau_a :: \kappa_a$$

$$\Delta; \Gamma \vdash e_f : \forall \alpha :: \kappa_a. \ \tau \quad \Delta; \Gamma \vdash e_f \ [\tau_a] : \tau_r[\tau_a/\alpha]$$

$$\Delta; \Gamma \vdash e : \tau \quad \tau \equiv \tau' \quad \Delta \vdash \tau' :: \star \quad \Delta; \Gamma \vdash e : \tau'$$

Syntax and type system easily extended with recursive and existential types.
Polymorphic List Library with higher-order $\exists$

List library is an existential package:

$$\text{pack}(\lambda \alpha::\star. \mu \xi::\star. \text{unit} + (\alpha \ast \xi), \text{list\_library})$$

as $\exists L::\star \Rightarrow \star. \{\text{empty} : \forall \alpha::\star. L \alpha;$

$\text{cons} : \forall \alpha::\star. \alpha \rightarrow L \alpha \rightarrow L \alpha;$

$\text{unlist} : \forall \alpha::\star. L \alpha \rightarrow \text{unit} + (\alpha \ast L \alpha);$

$\text{map} : \forall \alpha::\star. \forall \beta::\star. (\alpha \rightarrow \beta) \rightarrow L \alpha \rightarrow L \beta;$

$\ldots\}$$

The witness type operator is poly.lists: $\lambda \alpha::\star. \mu \xi::\star. \text{unit} + (\alpha \ast \xi)$.  

The existential type operator variable $L$ represents poly. lists.

List operations are polymorphic in element type.

The map function allows mapping $\alpha$ lists to $\beta$ lists.
Other Kinds of Kinds

Kinding systems for checking and tracking properties of type expressions:

- **Record kinds**
  - records at the type-level; define systems of mutually recursive types

- **Polymorphic kinds**
  - kind abstraction and application in types; System F “one level up”

- **Dependent kinds**
  - dependent types “one level up”

- **Row kinds**
  - describe “pieces” of record types for record polymorphism

- **Power kinds**
  - alternative presentation of subtyping

- **Singleton kinds**
  - formalize module systems with type sharing
Metatheory

System $F_\omega$ is type safe.
Metatheory

System $F_\omega$ is type safe.

▶ Preservation:
  Induction on typing derivation, using substitution lemmas:
  ▶ Term Substitution:
    if $\Delta_1, \Delta_2; \Gamma_1, x : \tau_x, \Gamma_2 \vdash e_1 : \tau$ and $\Delta_1; \Gamma_1 \vdash e_2 : \tau_x$, then $\Delta_1, \Delta_2; \Gamma_1, \Gamma_2 \vdash e_1[e_2/x] : \tau$.
  ▶ Type Substitution:
    if $\Delta_1, \alpha :: \kappa_\alpha, \Delta_2 \vdash \tau_1 :: \kappa$ and $\Delta_1 \vdash \tau_2 :: \kappa_\alpha$, then $\Delta_1, \Delta_2 \vdash \tau_1[\tau_2/\alpha] :: \kappa$.
  ▶ Type Substitution:
    if $\tau_1 \equiv \tau_2$, then $\tau_1[\tau/\alpha] \equiv \tau_2[\tau/\alpha]$.
  ▶ Type Substitution:
    if $\Delta_1, \alpha :: \kappa_\alpha, \Delta_2; \Gamma_1, \Gamma_2 \vdash e_1 : \tau$ and $\Delta_1 \vdash \tau_2 :: \kappa_\alpha$, then $\Delta_1, \Delta_2; \Gamma_1, \Gamma_2[\tau_2/\alpha] \vdash e_1[\tau_2/\alpha] : \tau[\tau_2/\alpha]$.
  ▶ All straightforward inductions, using various weakening and exchange lemmas.
Metatheory

System $\text{F}_\omega$ is type safe.

- Progress:
  Induction on typing derivation, using canonical form lemmas:
  - If $\cdot; \cdot \vdash v : \text{int}$, then $v = c$.
  - If $\cdot; \cdot \vdash v : \tau_a \rightarrow \tau_r$, then $v = \lambda x : \tau_a. \; e_b$.
  - If $\cdot; \cdot \vdash v : \forall \alpha::\kappa_a. \tau_r$, then $v = \Lambda \alpha::\kappa_a. \; e_b$.
  - Complicated by typing derivations that end with:

\[
\Delta; \Gamma \vdash e : \tau \quad \tau \equiv \tau' \quad \Delta \vdash \tau' :: \star \\
\Delta; \Gamma \vdash e : \tau'
\]

(just like with subtyping and subsumption).
Definitional Equivalence and Parallel Reduction

Parallel Reduction of $\tau$ to $\tau'$:

$$\tau \Rightarrow \tau'$$

$$\tau \Rightarrow \tau$$

$$\frac{\tau_{a1} \Rightarrow \tau_{a2} \quad \tau_{r1} \Rightarrow \tau_{r2}}{\tau_{a1} \rightarrow \tau_{r1} \Rightarrow \tau_{a2} \rightarrow \tau_{r2}}$$

$$\frac{\forall \alpha :: \kappa_a. \tau_{r1} \Rightarrow \forall \alpha :: \kappa_a. \tau_{r2}}{\forall \alpha :: \kappa_a. \tau_{r1} \Rightarrow \forall \alpha :: \kappa_a. \tau_{r2}}$$

$$\frac{\tau_{b1} \Rightarrow \tau_{b2}}{\lambda \alpha :: \kappa_a. \tau_{b1} \Rightarrow \lambda \alpha :: \kappa_a. \tau_{b2}}$$

$$\frac{\tau_{f1} \Rightarrow \tau_{f2} \quad \tau_{a1} \Rightarrow \tau_{a2}}{\tau_{f1} \tau_{a1} \Rightarrow \tau_{f2} \tau_{a2}}$$

$$\frac{\tau_{b} \Rightarrow \tau_{b}'}{\lambda \alpha :: \kappa_a. \tau_{b} \Rightarrow \tau_{b}'[\tau_{a}'/\alpha]}$$

A more “computational” relation.
Definitional Equivalence and Parallel Reduction

Key properties:

- Transitive and symmetric closure of parallel reduction and type equivalence coincide:
  \[
  \tau \leftrightarrow \tau' \iff \tau \equiv \tau'
  \]

- Parallel reduction has the Church-Rosser property:
  \[
  \text{If } \tau \xrightarrow{\ast} \tau_1 \text{ and } \tau \xrightarrow{\ast} \tau_2, \text{ then there exists } \tau' \text{ such that } \tau_1 \xrightarrow{\ast} \tau' \text{ and } \tau_2 \xrightarrow{\ast} \tau'.
  \]

- Equivalent types share a common reduct:
  \[
  \text{If } \tau_1 \equiv \tau_2, \text{ then there exists } \tau' \text{ such that } \tau_1 \xrightarrow{\ast} \tau' \text{ and } \tau_2 \xrightarrow{\ast} \tau'.
  \]

- Reduction preserves shapes:
  \[
  \text{If } \text{int} \xrightarrow{\ast} \tau', \text{ then } \tau' = \text{int}.
  \]
  \[
  \text{If } \tau_a \rightarrow \tau_r \xrightarrow{\ast} \tau', \text{ then } \tau' = \tau'_a \rightarrow \tau'_r \text{ and } \tau_a \xrightarrow{\ast} \tau'_a \text{ and } \tau_r \xrightarrow{\ast} \tau'_r.
  \]
  \[
  \text{If } \forall \alpha :: \kappa \cdot \tau_r \xrightarrow{\ast} \tau', \text{ then } \tau' = \forall \alpha :: \kappa \cdot \tau'_r \text{ and } \tau_r \xrightarrow{\ast} \tau'_r.
  \]
Definitional Equivalence and Parallel Reduction

Key properties:

- Transitive and symmetric closure of parallel reduction and type equivalence coincide:
  - $\tau \leftrightarrow^* \tau' \text{ iff } \tau \equiv \tau'$

- Parallel reduction has the Church-Rosser property:
  - If $\tau \leftrightarrow^* \tau_1$ and $\tau \leftrightarrow^* \tau_2$, then there exists $\tau'$ such that $\tau_1 \leftrightarrow^* \tau'$ and $\tau_2 \leftrightarrow^* \tau'$

- Equivalent types share a common reduct:
  - If $\tau_1 \equiv \tau_2$, then there exists $\tau'$ such that $\tau_1 \leftrightarrow^* \tau'$ and $\tau_2 \leftrightarrow^* \tau'$

- Reduction preserves shapes:
  - If $\text{int} \leftrightarrow^* \tau'$, then $\tau' = \text{int}$
  - If $\tau_a \rightarrow \tau_r \leftrightarrow^* \tau'$, then $\tau' = \tau_a \rightarrow \tau_r'$ and $\tau_a \leftrightarrow^* \tau_a'$ and $\tau_r \leftrightarrow^* \tau_r'$
Definitional Equivalence and Parallel Reduction

Key properties:

- Transitive and symmetric closure of parallel reduction and type equivalence coincide:
  \[ \tau \leftrightarrow^* \tau' \text{ iff } \tau \equiv \tau' \]

- Parallel reduction has the Church-Rosser property:
  If \( \tau \Rightarrow^* \tau_1 \) and \( \tau \Rightarrow^* \tau_2 \), then there exists \( \tau' \) such that \( \tau_1 \Rightarrow^* \tau' \) and \( \tau_2 \Rightarrow^* \tau' \)
Definitional Equivalence and Parallel Reduction

Key properties:

- Transitive and symmetric closure of parallel reduction and type equivalence coincide:
  \[ \tau \Leftrightarrow^* \tau' \text{ iff } \tau \equiv \tau' \]

- Parallel reduction has the Church-Rosser property:
  - If \( \tau \Rightarrow^* \tau_1 \) and \( \tau \Rightarrow^* \tau_2 \), then there exists \( \tau' \) such that \( \tau_1 \Rightarrow^* \tau' \) and \( \tau_2 \Rightarrow^* \tau' \)

- Equivalent types share a common reduct:
  - If \( \tau_1 \equiv \tau_2 \), then there exists \( \tau' \) such that \( \tau_1 \Rightarrow^* \tau' \) and \( \tau_2 \Rightarrow^* \tau' \)
Definitional Equivalence and Parallel Reduction

Key properties:

- Transitive and symmetric closure of parallel reduction and type equivalence coincide:
  \[ \tau \leftrightarrow* \tau' \text{ iff } \tau \equiv \tau' \]

- Parallel reduction has the Church-Rosser property:
  If \( \tau \Rightarrow* \tau_1 \) and \( \tau \Rightarrow* \tau_2 \),
  then there exists \( \tau' \) such that \( \tau_1 \Rightarrow* \tau' \) and \( \tau_2 \Rightarrow* \tau' \)

- Equivalent types share a common reduct:
  If \( \tau_1 \equiv \tau_2 \), then there exists \( \tau' \) such that \( \tau_1 \Rightarrow* \tau' \) and \( \tau_2 \Rightarrow* \tau' \)

- Reduction preserves shapes:
  
  - If \( \text{int} \Rightarrow* \tau' \), then \( \tau' = \text{int} \)
  
  - If \( \tau_a \rightarrow \tau_r \Rightarrow* \tau' \), then \( \tau' = \tau_a' \rightarrow \tau_r' \) and \( \tau_a \Rightarrow* \tau_a' \) and \( \tau_r \Rightarrow* \tau_r' \)
  
  - If \( \forall \alpha :: \kappa_a \cdot \tau_r \Rightarrow* \tau' \), then \( \tau' = \forall \alpha :: \kappa_a \cdot \tau_r' \) and \( \tau_r \Rightarrow* \tau_r' \)
Canonical Forms

If \( \cdot ; \cdot \vdash v : \tau_a \rightarrow \tau_r \), then \( v = \lambda x : \tau_a . \ e_b \).

Proof:
By cases on the form of \( v \):
Canonical Forms

If \( \vdash v : \tau_a \rightarrow \tau_r \), then \( v = \lambda x:\tau_a. \ e_b \).

Proof:

By cases on the form of \( v \):

\( v = \lambda x:\tau_a. \ e_b \).

We have that \( v = \lambda x:\tau_a. \ e_b \).
Canonical Forms

If \( \cdot; \cdot \vdash v : \tau_a \rightarrow \tau_r \), then \( v = \lambda x : \tau_a . \; e_b \).

Proof:

By cases on the form of \( v \):

1. \( v = c \).

Derivation of \( \cdot; \cdot \vdash v : \tau_a \rightarrow \tau_r \) must be of the form:

\[
\cdot; \cdot \vdash c : \text{int} \quad \text{int} \equiv \tau_1 \\
\cdot; \cdot \vdash c : \tau_1 \\
\cdot; \cdot \vdash c : \tau_{n-1} \quad \tau_{n-1} \equiv \tau_n \\
\cdot; \cdot \vdash c : \tau_n \\
\cdot; \cdot \vdash c : \tau_a \rightarrow \tau_r \\
\]

Therefore, we can construct the derivation \( \text{int} \equiv \tau_a \rightarrow \tau_r \).

We can find a common reduct: \( \text{int} \Rightarrow^* \tau^\uparrow \) and \( \tau_a \rightarrow \tau_r \Rightarrow^* \tau^\uparrow \).

Reduction preserves shape: \( \text{int} \Rightarrow^* \tau^\uparrow \) implies \( \tau^\uparrow = \text{int} \).

Reduction preserves shape: \( \tau_a \rightarrow \tau_r \Rightarrow^* \tau^\uparrow \) implies \( \tau^\uparrow = \tau_a' \rightarrow \tau_r' \).

But, \( \tau^\uparrow = \text{int} \) and \( \tau^\uparrow = \tau_a' \rightarrow \tau_r' \) is a contradiction.
Canonical Forms

If $\cdot; \cdot \vdash v : \tau_a \rightarrow \tau_r$, then $v = \lambda x : \tau_a.\ e_b$.

Proof:
By cases on the form of $v$:

1. $v = \Lambda \alpha :: \kappa_a.\ e_b$.
   Derivation of $\cdot; \cdot \vdash v : \tau_a \rightarrow \tau_r$ must be of the form:

   $\vdash \Lambda \alpha :: \kappa_a.\ e_b : \forall \alpha :: \kappa_a.\ \tau_z \quad \forall \alpha :: \kappa_a.\ \tau_z \equiv \tau_1$
   $\vdash \Lambda \alpha :: \kappa_a.\ e_b : \tau_1$

   $\vdash \Lambda \alpha :: \kappa_a.\ e_b : \tau_{n-1} \equiv \tau_n$

   $\vdash \Lambda \alpha :: \kappa_a.\ e_b : \tau_n \equiv \tau_a \rightarrow \tau_r$

Therefore, we can construct the derivation $\forall \alpha :: \kappa_a.\ \tau_z \equiv \tau_a \rightarrow \tau_r$.

We can find a common reduct: $\forall \alpha :: \kappa_a.\ \tau_z \Rightarrow^* \tau^\dagger$ and $\tau_a \rightarrow \tau_r \Rightarrow^* \tau^\dagger$.

Reduction preserves shape: $\forall \alpha :: \kappa_a.\ \tau_z \Rightarrow^* \tau^\dagger$ implies $\tau^\dagger = \forall \alpha :: \kappa_a.\ \tau'_z$.

Reduction preserves shape: $\tau_a \rightarrow \tau_r \Rightarrow^* \tau^\dagger$ implies $\tau^\dagger = \tau'_a \rightarrow \tau'_r$.

But, $\tau^\dagger = \forall \alpha :: \kappa_a.\ \tau'_z$ and $\tau^\dagger = \tau'_a \rightarrow \tau'_r$ is a contradiction.
Metatheory

System $F_\omega$ is type safe.

Where was the $\Delta \vdash \tau :: \kappa$ judgement used in the proof?
Metatheory

System $F_\omega$ is type safe.

Where was the $\Delta \vdash \tau :: \kappa$ judgement used in the proof? In Type Substitution lemmas, but only in an inessential way.
Metatheory

System $F_\omega$ is type safe.

Where was the $\Delta \vdash \tau :: \kappa$ judgement used in the proof? In Type Substitution lemmas, but only in an inessential way.

After weeks of thinking about type systems, kinding seems natural; but kinding is not required for type safety!
System $F_\omega$ without Kinds / System F with Type-Level Abstraction and Application

\[
e ::= c \mid x \mid \lambda x : \tau. \ e \mid e \ e \mid \Lambda \alpha. \ e \mid e \ [\tau] \\
\nu ::= c \mid \lambda x : \tau. \ e \mid \Lambda \alpha. \ e \\
\tau ::= \text{int} \mid \tau \rightarrow \tau \mid \alpha \mid \forall \alpha. \ \tau \mid \lambda \alpha. \ \tau \mid \tau \ \tau \\
\Gamma ::= \cdot \mid \Gamma, x : \tau \\
\Delta ::= \cdot \mid \Delta, \alpha
\]
System F_ω without Kinds / System F with Type-Level Abstraction and Application

\[ e ::= c \mid x \mid \lambda x: \tau. \ e \mid e \ e \mid \Lambda \alpha. \ e \mid e \ [\tau] \]

\[ v ::= c \mid \lambda x: \tau. \ e \mid \Lambda \alpha. \ e \]

\[ \tau ::= \text{int} \mid \tau \rightarrow \tau \mid \alpha \mid \forall \alpha. \ \tau \mid \lambda \alpha. \ \tau \mid \tau \ \tau \]

\[ \Gamma ::= \cdot \mid \Gamma, \ x: \tau \]

\[ \Delta ::= \cdot \mid \Delta, \ \alpha \]

\[ e \rightarrow_{\text{cbv}} e' \]

\[
\frac{}{(\lambda x:\tau. \ e_b) \ v_a} \rightarrow_{\text{cbv}} e_b[v_a/x]
\]

\[
\frac{}{e_f \rightarrow_{\text{cbv}} e'_f}
\]

\[
\frac{e_f \ e_a} {e_f \ e_a \rightarrow_{\text{cbv}} e'_f \ e_a}
\]

\[
\frac{}{v_f \ e_a \rightarrow_{\text{cbv}} v_f \ e'_a}
\]

\[
\frac{}{(\Lambda \alpha. \ e_b) \ [\tau_a]} \rightarrow_{\text{cbv}} e_b[\tau_a/\alpha]
\]

\[
\frac{}{e_f \rightarrow_{\text{cbv}} e'_f}
\]

\[
\frac{e_f \ [\tau_a]} {e_f \ [\tau_a] \rightarrow_{\text{cbv}} e'_f \ [\tau_a]}
\]
System $F_\omega$ without Kinds / System F with Type-Level Abstraction and Application

\[
e ::= c \mid x \mid \lambda x:\tau. \ e \mid e \ e \mid \Lambda \alpha. \ e \mid e [\tau]
\]
\[
v ::= c \mid \lambda x:\tau. \ e \mid \Lambda. \ e
\]
\[
\tau ::= \text{int} \mid \tau \rightarrow \tau \mid \alpha \mid \forall \alpha. \ \tau \mid \lambda \alpha. \ \tau \mid \tau \ \tau
\]
\[
\Gamma ::= \cdot \mid \Gamma, x:\tau
\]
\[
\Delta ::= \cdot \mid \Delta, \alpha
\]

Check that free type variables of $\tau$ are in $\Delta$, but nothing else.
System $F_\omega$ without Kinds / System F with Type-Level Abstraction and Application

\[
\begin{align*}
\text{e} & ::= c \mid x \mid \lambda x: \tau.\ e \mid e\ e \mid \Lambda\alpha.\ e \mid e[\tau] \\
\text{v} & ::= c \mid \lambda x: \tau.\ e \mid \Lambda\alpha.\ e \\
\tau & ::= \text{int} \mid \tau \rightarrow \tau \mid \alpha \mid \forall\alpha.\ \tau \mid \lambda\alpha.\ \tau \mid \tau\ \tau \\
\end{align*}
\]

\[\tau \equiv \tau'\]

\[
\begin{align*}
\frac{\tau \equiv \tau}{\frac{\tau_1 \equiv \tau_2}{\tau_1 \equiv \tau_2}} & \quad \frac{\tau_2 \equiv \tau_1}{\tau_2 \equiv \tau_1} \\
\frac{\tau_a \equiv \tau_{a_2}}{	au_a \rightarrow \tau_{r_1} \equiv \tau_{a_2} \rightarrow \tau_{r_2}} & \quad \frac{\tau_{r_1} \equiv \tau_{r_2}}{orall\alpha.\ \tau_{r_1} \equiv \forall\alpha.\ \tau_{r_2}} \\
\frac{\tau_{b_1} \equiv \tau_{b_2}}{\lambda\alpha.\ \tau_{b_1} \equiv \lambda\alpha.\ \tau_{b_2}} & \quad \frac{\tau_{f_1} \equiv \tau_{f_2}}{\frac{\tau_{a_1} \equiv \tau_{a_2}}{\tau_{f_1} \tau_{a_1} \equiv \tau_{f_2} \tau_{a_2}}} \\
\frac{(\lambda\alpha.\ \tau_b)\ \tau_a \equiv \tau_b[\tau_a/\alpha]}{	au_{f_1} \tau_{a_1} \equiv \tau_{f_2} \tau_{a_2}}
\end{align*}
\]
System $F_\omega$ without Kinds / System F with Type-Level Abstraction and Application

\[
e ::= \ c \mid x \mid \lambda x: \tau. \ e \mid e \ e \mid \Lambda \alpha. \ e \mid e [\tau]
\]

\[
\nu ::= \ c \mid \lambda x: \tau. \ e \mid \Lambda \alpha. \ e
\]

\[
\tau ::= \ \text{int} \mid \tau \rightarrow \tau \mid \alpha \mid \forall \alpha. \ \tau \mid \lambda \alpha. \ \tau \mid \tau \ \tau
\]

\[
\Gamma ::= \ \cdot \mid \Gamma, x: \tau
\]

\[
\Delta ::= \ \cdot \mid \Delta, \alpha
\]

\[
\Delta; \Gamma \vdash e : \tau
\]

\[
\frac{\Delta \vdash \tau_a :: \checkmark}{\Delta; \Gamma \vdash \lambda x: \tau_a. \ e_b : \tau_a \rightarrow \tau_r}
\]

\[
\frac{\Delta \vdash \tau_a :: \checkmark}{\Delta; \Gamma \vdash \lambda x: \tau_a. \ e_b : \forall \alpha. \ \tau_r}
\]

\[
\frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau}
\]

\[
\frac{\Delta; \Gamma \vdash e_f : \tau_a \rightarrow \tau_r}{\Delta; \Gamma \vdash e_f \ e_a : \tau_r}
\]

\[
\frac{\Delta; \Gamma \vdash e_f : \forall \alpha. \ \tau_r}{\Delta; \Gamma \vdash e_f [\tau_a] : \tau_r[\tau_a / \alpha]}
\]

\[
\frac{\Delta; \Gamma \vdash \tau \equiv \tau'}{\Delta; \Gamma \vdash \tau' :: \checkmark}
\]

\[
\Delta; \Gamma \vdash e : \tau'
\]
System $F_\omega$ without Kinds / System F with Type-Level Abstraction and Application

This language is type safe.
System $F_\omega$ without Kinds / System F with Type-Level Abstraction and Application

This language is type safe.

- **Preservation:**
  - Induction on typing derivation, using substitution lemmas:
    - **Term Substitution:**
      
      \[
      \text{if } \Delta_1, \Delta_2; \Gamma_1, x : \tau_x, \Gamma_2 \vdash e_1 : \tau \text{ and } \Delta_1; \Gamma_1 \vdash e_2 : \tau_x, \\
      \text{then } \Delta_1, \Delta_2; \Gamma_1, \Gamma_2 \vdash e_1[e_2/x] : \tau.
      \]
    - **Type Substitution:**
      
      \[
      \text{if } \Delta_1, \alpha, \Delta_2 \vdash \tau_1 :: \checkmark \text{ and } \Delta_1 \vdash \tau_2 :: \checkmark, \\
      \text{then } \Delta_1, \Delta_2 \vdash \tau_1[\tau_2/\alpha] :: \checkmark.
      \]
    - **Type Substitution:**
      
      \[
      \text{if } \tau_1 \equiv \tau_2, \text{ then } \tau_1[\tau/\alpha] \equiv \tau_2[\tau/\alpha].
      \]
    - **Type Substitution:**
      
      \[
      \text{if } \Delta_1, \alpha, \Delta_2; \Gamma_1, \Gamma_2 \vdash e_1 : \tau \text{ and } \Delta_1 \vdash \tau_2 :: \checkmark, \\
      \text{then } \Delta_1, \Delta_2; \Gamma_1, \Gamma_2[\tau_2/\alpha] \vdash e_1[\tau_2/\alpha] : \tau[\tau_2/\alpha].
      \]
    - **All straightforward inductions, using various weakening and exchange lemmas.**
This language is type safe.

- **Progress:**
  - Induction on typing derivation, using canonical form lemmas:
    - If $\cdot;\cdot \vdash v : \text{int}$, then $v = c$.
    - If $\cdot;\cdot \vdash v : \tau_a \rightarrow \tau_r$, then $v = \lambda x:\tau_a.\ e_b$.
    - If $\cdot;\cdot \vdash v : \forall \alpha.\ \tau_r$, then $v = \Lambda \alpha.\ e_b$.

- Using parallel reduction relation.
Why Kinds?

Why aren’t kinds required for type safety?
Why Kinds?

Why aren’t kinds required for type safety?

Recall statement of type safety:

\[ \text{If } \cdot; \cdot \vdash e : \tau, \text{ then } e \text{ does not get stuck.} \]
Why Kinds?

Why aren’t kinds required for type safety?

Recall statement of type safety:

\[
\text{If } \cdot; \cdot \vdash e : \tau, \text{ then } e \text{ does not get stuck.}
\]

The typing derivation \( \cdot;\cdot \vdash e : \tau \) includes definitional-equivalence sub-derivations \( \tau \equiv \tau' \), which are explicit evidence that \( \tau \) and \( \tau' \) are the same.

▶ E.g., to show that the “natural” type of the function expression in an application is equivalent to an arrow type:

\[
\begin{align*}
\Delta; \Gamma, \cdot \vdash e_f : \tau_f & \quad \tau_f \equiv \tau_a \rightarrow \tau_r \quad \tau_r \equiv \tau_f' \\
\Delta; \Gamma \vdash e_f : \tau_a \rightarrow \tau_r & \quad \Delta; \Gamma \vdash e_a : \tau_a \quad \Delta; \Gamma \vdash e_f e_a : \tau_r
\end{align*}
\]
Why Kinds?

Why aren’t kinds required for type safety?

Recall statement of type safety:

If $\cdot; \cdot \vdash e : \tau$, then $e$ does not get stuck.

The typing derivation $\cdot; \cdot \vdash e : \tau$
includes definitional-equivalence sub-derivations $\tau \equiv \tau'$,
which are explicit evidence that $\tau$ and $\tau'$ are the same.

Definitional equivalence ($\tau \equiv \tau'$) and parallel reduction ($\tau \Rightarrow \tau'$)
do not require well-kinded types
(although they preserve the kinds of well-kinded types).

- E.g., $(\lambda \alpha. \alpha \rightarrow \alpha) \ (\text{int int}) \equiv (\text{int int}) \rightarrow (\text{int int})$
Why Kinds?

Why aren’t kinds required for type safety?

Recall statement of type safety:

\[
\text{If } \cdot; \cdot \vdash e : \tau, \text{ then } e \text{ does not get stuck.}
\]

The typing derivation \( \cdot; \cdot \vdash e : \tau \)
includes definitional-equivalence sub-derivations \( \tau \equiv \tau' \),
which are explicit evidence that \( \tau \) and \( \tau' \) are the same.

Definitional equivalence \((\tau \equiv \tau')\) and parallel reduction \((\tau \Rightarrow \tau')\)
do not require well-kinded types
(although they preserve the kinds of well-kinded types).

Type (and kind) erasure means that “wrong/bad/meaningless” types
do not affect run-time behavior.

- Ill-kinded types can’t make well-typed terms get stuck.
Why Kinds?

Kinds aren’t for *type safety*:

- Because a typing derivation (even with ill-kindred types), carries enough evidence to guarantee that expressions don’t get stuck.
Why Kinds?

Kinds aren’t for *type safety*:
▶ Because a typing derivation (even with ill-kindied types),
   carries enough evidence to guarantee that expressions don’t get stuck.

Kinds are for *type checking*:
▶ Because programmers write programs, not typing derivations.
▶ Because type checkers are algorithms.
Why Kinds?

Kinds are for *type checking*:

- Because programmers write programs, not typing derivations.
- Because type checkers are algorithms.

Recall the statement of type checking:

\[ \Delta; \Gamma \vdash e : \tau \]

Two issues:

- \[ \Delta; \Gamma \vdash e : \tau \] and \[ \tau \equiv \tau' \] is a non-syntax-directed rule
- \[ \Delta; \Gamma \vdash e : \tau \] is a non-syntax-directed relation

One non-issue:

- \[ \Delta \vdash \tau :: \kappa \] is a syntax-directed relation (STLC "one level up")

Matthew Fluet
Programming Language Theory
Lecture 20
Why Kinds?

Kinds are for *type checking*:

- Because programmers write programs, not typing derivations.
- Because type checkers are algorithms.

Recall the statement of type checking:

\[
\text{Given } \Delta, \Gamma, \text{ and } e, \text{ does there exist } \tau \text{ such that } \Delta; \Gamma \vdash e : \tau.\]

Two issues:

- \(\Delta; \Gamma \vdash e : \tau \equiv \tau'\)
  - \(\Delta \vdash \tau' :: \kappa\) is a non-syntax-directed rule

- \(\tau \equiv \tau'\) is a non-syntax-directed relation

One non-issue:

- \(\Delta \vdash \tau :: \kappa\) is a syntax-directed relation (STLC “one level up”)
Why Kinds?

Kinds are for type checking:

▶ Because programmers write programs, not typing derivations.
▶ Because type checkers are algorithms.

Recall the statement of type checking:

Given $\Delta$, $\Gamma$, and $e$, does there exist $\tau$ such that $\Delta; \Gamma \vdash e : \tau$.

Two issues:

▶ $\Delta; \Gamma \vdash e : \tau \quad \tau \equiv \tau' \quad \Delta \vdash \tau' :: \star$ is a non-syntax-directed rule
▶ $\tau \equiv \tau'$ is a non-syntax-directed relation

One non-issue:

▶ $\Delta \vdash \tau :: \kappa$ is a syntax-directed relation (STLC “one level up”)
Type Checking for System $F_\omega$

Remove non-syntax-directed rules and relations:

\[
\Delta; \Gamma \vdash e : \tau
\]

\[
\Delta; \Gamma \vdash c : \text{int}
\]

\[
\Delta; \Gamma \vdash \lambda x : \tau_a. e_b : \tau_a \rightarrow \tau_r
\]

\[
\Gamma(x) = \tau
\]

\[
\Delta; \Gamma \vdash x : \tau
\]

\[
\Delta, \alpha :: \kappa_a; \Gamma \vdash e_b : \tau_r
\]

\[
\Delta; \Gamma \vdash \land \alpha. e_b : \land \alpha :: \kappa_a. \tau_r
\]

\[
\Delta; \Gamma \vdash e_f : \tau_f
\]

\[
\tau_f \Rightarrow \tau'_f
\]

\[
\tau'_f = \tau'_{fa} \rightarrow \tau'_{fr}
\]

\[
\Delta; \Gamma \vdash e_a : \tau_a
\]

\[
\tau_a \Rightarrow \tau'_a
\]

\[
\tau'_{fa} = \tau'_{a}
\]

\[
\Delta; \Gamma \vdash e_f, e_a : \tau'_{fr}
\]

\[
\Delta; \Gamma \vdash e_f : \tau_f
\]

\[
\tau_f \Rightarrow \tau'_f
\]

\[
\tau'_f = \land \alpha :: \kappa_{fa}. \tau_{fr}
\]

\[
\Delta \vdash \tau_a :: \kappa_a
\]

\[
\kappa_{fa} = \kappa_a
\]

\[
\Delta; \Gamma \vdash e_f [\tau_a] : \tau_{fr}[\tau_a/\alpha]
\]
Type Checking for System $F_\omega$

Kinds are for *type checking*.

Given $\Delta$, $\Gamma$, and $e$, does there exist $\tau$ such that $\Delta; \Gamma \vdash e : \tau$.

Metatheory for kind system:

- Well-kinded types don't get stuck.
- If $\Delta \vdash \tau :: \kappa$ and $\tau \mapsto \tau'$, then either $\tau'$ is in (weak-head) normal form (i.e., a type-level "value"
- or $\tau' \mapsto \tau''$.
- But, irrelevant for type checking of expressions.
- Well-kinded types terminate.
- If $\Delta \vdash \tau :: \kappa$, then there exists $\tau'$ such that $\tau \mapsto \tau'$.
- Proof is similar to that of termination of STLC.

Type checking for System $F_\omega$ is decidable.
Type Checking for System $F_\omega$

Kinds are for *type checking*.

Given $\Delta$, $\Gamma$, and $e$, does there exist $\tau$ such that $\Delta; \Gamma \vdash e : \tau$.

Metatheory for kind system:

- Well-kindled types don’t get stuck.
  - If $\Delta \vdash \tau :: \kappa$ and $\tau \Rightarrow^* \tau'$, then either $\tau'$ is in (weak-head) normal form (i.e., a type-level “value”) or $\tau' \Rightarrow \tau''$.
  - Proof by Progress and Preservation on kinding and parallel reduction derivations.
Type Checking for System $\text{F}_\omega$

Kinds are for type checking.

Given $\Delta$, $\Gamma$, and $e$, does there exist $\tau$ such that $\Delta; \Gamma \vdash e : \tau$.

Metatheory for kind system:

▶ Well-kinded types don’t get stuck.
  ▶ If $\Delta \vdash \tau :: \kappa$ and $\tau \Rightarrow^* \tau'$, then either $\tau'$ is in (weak-head) normal form (i.e., a type-level “value”) or $\tau' \Rightarrow \tau''$.
  ▶ Proof by Progress and Preservation on kinding and parallel reduction derivations.
▶ But, irrelevant for type checking of expressions.
  If $\tau_f \Rightarrow^* \tau'_f$ “gets stuck” at a type $\tau'_f$ that is not an arrow type, then the application typing rule does not apply and a typing derivation does not exist.
Type Checking for System $F_\omega$

Kinds are for *type checking*.

Given $\Delta$, $\Gamma$, and $e$, does there exist $\tau$ such that $\Delta; \Gamma \vdash e : \tau$.

Metatheory for kind system:

- Well-kinded types don’t get stuck.
  - If $\Delta \vdash \tau :: \kappa$ and $\tau \Rightarrow^* \tau'$, then either $\tau'$ is in (weak-head) normal form (i.e., a type-level “value”) or $\tau' \Rightarrow \tau''$.
  - But, irrelevant for type checking of expressions.
Type Checking for System F_ω

Kinds are for type checking.

Given Δ, Γ, and e, does there exist τ such that Δ; Γ ⊢ e : τ.

Metatheory for kind system:

- Well-kinded types don’t get stuck.
  - If Δ ⊢ τ :: κ and τ ⇒* τ’, then either τ’ is in (weak-head) normal form (i.e., a type-level “value”) or τ’ ⇒ τ”.
  - But, irrelevant for type checking of expressions.

- Well-kinded types terminate.
  - If Δ ⊢ τ :: κ, then there exists τ’ such that τ ⇒↓ τ’.
  - Proof is similar to that of termination of STLC.
Type Checking for System $F_\omega$

Kinds are for *type checking*.

Given $\Delta$, $\Gamma$, and $e$, does there exist $\tau$ such that $\Delta; \Gamma \vdash e : \tau$.

Metatheory for kind system:

▶ Well-kindred types don’t get stuck.
  ▶ If $\Delta \vdash \tau :: \kappa$ and $\tau \Rightarrow^* \tau'$, then either $\tau'$ is in (weak-head) normal form (i.e., a type-level “value”) or $\tau' \Rightarrow \tau''$.
  ▶ But, irrelevant for type checking of expressions.

▶ Well-kindred types *terminate*.
  ▶ If $\Delta \vdash \tau :: \kappa$, then there exists $\tau'$ such that $\tau \Downarrow \tau'$.
  ▶ Proof is similar to that of termination of STLC.

Type checking for System $F_\omega$ is decidable.
Going Further

This is just the tip of an iceberg.

- Pure type systems
  - Why stop at three levels of expressions (terms, types, and kinds)?
  - Allow abstraction and application at the level of kinds, and introduce sorts to classify kinds.
  - Why stop at four levels of expressions?
  - ... 
  - “For programming languages, however, three levels have proved sufficient.”