CSE-505: Programming Languages

Lecture 27 — Higher-Order Polymorphism

Matthew Fluet
2015

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

$$\begin{array}{rcl} e & ::= & c \mid x \mid \lambda x{:}\tau.\ e \mid e\ e \mid \\ & & \Lambda\alpha.\ e \mid e\ [\tau] \mid \\ & & \mathtt{pack}_{\exists\alpha.\ \tau}(\tau, e) \mid \mathtt{unpack}\ e\ \mathtt{as}\ (\alpha, x)\ \mathtt{in}\ e \mid \\ & & \mathtt{roll}_{\mu\alpha.\ \tau}(e) \mid \mathtt{unroll}(e) \\ v & ::= & c \mid \lambda x{:}\tau.\ e \mid \Lambda\alpha.\ e \mid \mathtt{pack}_{\exists\alpha.\ \tau}(\tau, v) \mid \mathtt{roll}_{\mu\alpha.\ \tau}(v) \end{array}$$

$\boxed{e \to_{\mathsf{cbv}} e'}$

$$\frac{}{(\lambda x{:}\ \tau.\ e_b)\ v_a \to_{\mathsf{cbv}} e_b[v_a/x]} \qquad \frac{e_f \to_{\mathsf{cbv}} e_f'}{e_f\ e_a \to_{\mathsf{cbv}} e_f'\ e_a} \qquad \frac{e_a \to_{\mathsf{cbv}} e_a'}{v_f\ e_a \to_{\mathsf{cbv}} v_f\ e_a'}$$

$$\frac{}{(\Lambda\alpha.\ e_b)\ [\tau_a] \to_{\mathsf{cbv}} e_b[\tau_a/\alpha]} \qquad \frac{e_f \to_{\mathsf{cbv}} e_f'}{e_f\ [\tau_a] \to_{\mathsf{cbv}} e_f'\ [\tau_a]}$$

$$\frac{e_a \to_{\mathsf{cbv}} e_a'}{\mathtt{pack}_{\exists\alpha.\ \tau}(\tau_w, e_a) \to_{\mathsf{cbv}} \mathtt{pack}_{\exists\alpha.\ \tau}(\tau_w, e_a')}$$

$$\frac{e_a \to_{\mathsf{cbv}} e_a'}{\mathtt{unpack}\ e_a\ \mathtt{as}\ (\alpha, x)\ \mathtt{in}\ e_b \to_{\mathsf{cbv}} \mathtt{unpack}\ e_a'\ \mathtt{as}\ (\alpha, x)\ \mathtt{in}\ e_b}$$

$$\frac{}{\mathtt{unpack}\ \mathtt{pack}_{\exists\alpha.\ \tau}(\tau_w, v_a)\ \mathtt{as}\ (\alpha, x)\ \mathtt{in}\ e_b \to_{\mathsf{cbv}} e_b[\tau_w/\alpha][v_a/x]}$$

$$\frac{e_a \to_{\mathsf{cbv}} e_a'}{\mathtt{unroll}(e_a) \to_{\mathsf{cbv}} \mathtt{unroll}(e_a')} \qquad \frac{}{\mathtt{unroll}(\mathtt{roll}_{\mu\alpha.\ \tau}(v_a)) \to_{\mathsf{cbv}} v_a}$$

System F with Recursive and Existential Types

$$\begin{array}{rcl} \tau & ::= & \mathsf{int} \mid \tau \to \tau \mid \alpha \mid \forall \alpha.\ \tau \mid \exists \alpha.\ \tau \mid \mu \alpha.\ \tau \\ \Delta & ::= & \cdot \mid \Delta, \alpha \\ \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \end{array}$$

$\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\frac{}{\Delta; \Gamma \vdash c : \mathsf{int}} \qquad \frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau}$$

$$\frac{\Delta \vdash \tau_a \qquad \Delta; \Gamma, x : \tau_a \vdash e_b : \tau_r}{\Delta; \Gamma \vdash \lambda x{:}\tau_a.\ e_b : \tau_a \to \tau_r} \qquad \frac{\Delta; \Gamma \vdash e_f : \tau_a \to \tau_r \qquad \Delta; \Gamma \vdash e_a : \tau_a}{\Delta; \Gamma \vdash e_f\ e_a : \tau_r}$$

$$\frac{\Delta, \alpha; \Gamma \vdash e_b : \tau_r}{\Delta; \Gamma \vdash \Lambda \alpha.\ e_b : \forall \alpha.\ \tau_r} \qquad \frac{\Delta; \Gamma \vdash e_f : \forall \alpha.\ \tau_r \qquad \Delta \vdash \tau_a}{\Delta; \Gamma \vdash e_f\ [\tau_a] : \tau_r[\tau_a/\alpha]}$$

$$\frac{\Delta; \Gamma \vdash e_a : \tau[\tau_w/\alpha]}{\Delta; \Gamma \vdash \mathtt{pack}_{\exists \alpha.\ \tau}(\tau_w, e_a) : \exists \alpha.\ \tau} \qquad \frac{\Delta; \Gamma \vdash e_a : \exists \alpha.\ \tau \qquad \Delta, \alpha; \Gamma, x{:}\tau \vdash e_b : \tau_r \qquad \Delta \vdash \tau_r}{\Delta; \Gamma \vdash \mathtt{unpack}\ e_a\ \mathtt{as}\ (\alpha, x)\ \mathtt{in}\ e_b : \tau_r}$$

$$\frac{\Delta; \Gamma \vdash e_a : \tau[(\mu \alpha.\ \tau)/\alpha]}{\Delta; \Gamma \vdash \mathtt{roll}_{\mu \alpha.\ \tau}(e_a) : \mu \alpha.\ \tau} \qquad \frac{\Delta; \Gamma \vdash e_a : \mu \alpha.\ \tau}{\Delta; \Gamma \vdash \mathtt{unroll}(e_a) : \tau[(\mu \alpha.\ \tau)/\alpha]}$$

Goal

Understand what this interface means and why it matters:

```
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:

$$\begin{array}{l}\texttt{pack}(\mu\xi.\, \textbf{unit} + (\textbf{int} * \xi), \mathit{list\_library}) \\ \texttt{as}\ \exists L.\ \{\textbf{empty} : L; \\ \qquad\qquad \textbf{cons} : \textbf{int} \to L \to L; \\ \qquad\qquad \textbf{unlist} : L \to \textbf{unit} + (\textbf{int} * L); \\ \qquad\qquad \textbf{map} : (\textbf{int} \to \textbf{int}) \to L \to L; \\ \qquad\qquad \ldots\}\end{array}$$

The witness type is integer lists: $\mu\xi.\, \textbf{unit} + (\textbf{int} * \xi)$.

The existential type variable L represents integer lists.

List operations are monomorphic in element type (**int**).

The **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:

$$
\begin{array}{l}
\Lambda\alpha.\ \texttt{pack}(\mu\xi.\ \textbf{unit} + (\alpha * \xi), \textit{list\_library}) \\
\quad \texttt{as}\ \exists L.\ \{\textbf{empty} : L; \\
\qquad\qquad \textbf{cons} : \alpha \to L \to L; \\
\qquad\qquad \textbf{unlist} : L \to \textbf{unit} + (\alpha * L); \\
\qquad\qquad \textbf{map} : (\alpha \to \alpha) \to L \to L; \\
\qquad\qquad \ldots\}
\end{array}
$$

The witness type is α lists: $\mu\xi.\ \textbf{unit} + (\alpha * \xi)$.

The existential type variable L represents α lists.

List operations are monomorphic in element type (α).

The **map** function only allows mapping α lists to α lists.

Type Abbreviations and Type Operators

Reasonable enough to provide list type as a *(parametric) type abbreviation*:

$$\mathbf{L}\ \alpha \ = \ \mu\xi.\ \textbf{unit} + (\alpha * \xi)$$

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

Gives an *informal* notion of functions at the type-level.

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

- How "modular" and "safe" are libraries built from cpp macros?

Type Abbreviations and Type Operators

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

- a function from types to types

$$\mathbf{L} \;=\; \boldsymbol{\lambda\alpha.\ \mu\xi.}\ \textbf{unit} + (\boldsymbol{\alpha} * \boldsymbol{\xi})$$

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.

$$\mathbf{Id} \;=\; \lambda\alpha.\,\alpha$$

$$\mathbf{int} \rightarrow \mathbf{bool} \qquad \mathbf{int} \rightarrow \mathbf{Id}\ \mathbf{bool} \qquad \mathbf{Id}\ \mathbf{int} \rightarrow \mathbf{bool} \qquad \mathbf{Id}\ \mathbf{int} \rightarrow \mathbf{Id}\ \mathbf{bool}$$

$$\mathbf{Id}\ (\mathbf{int} \rightarrow \mathbf{bool}) \qquad \mathbf{Id}\ (\mathbf{Id}\ (\mathbf{int} \rightarrow \mathbf{bool})) \qquad \ldots$$

Type-level Expressions

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

$$\mathbf{Id} \;=\; \lambda\alpha.\,\alpha$$

$\mathbf{int} \to \mathbf{bool}$ $\quad$ $\mathbf{int} \to \mathbf{Id}\ \mathbf{bool}$ $\quad$ $\mathbf{Id}\ \mathbf{int} \to \mathbf{bool}$ $\quad$ $\mathbf{Id}\ \mathbf{int} \to \mathbf{Id}\ \mathbf{bool}$

$\mathbf{Id}\ (\mathbf{int} \to \mathbf{bool})$ $\quad$ $\mathbf{Id}\ (\mathbf{Id}\ (\mathbf{int} \to \mathbf{bool}))$ $\quad$ $\ldots$

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

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

$$\ldots \qquad \frac{}{(\lambda\alpha.\,\tau_b)\ \tau_a \equiv \tau_b[\alpha/\tau_a]} \qquad \ldots$$

Type-level Expressions

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

$$\textbf{Id} \;=\; \lambda\alpha.\, \alpha$$

$$\textbf{int} \to \textbf{bool} \qquad \textbf{int} \to \textbf{Id bool} \qquad \textbf{Id int} \to \textbf{bool} \qquad \textbf{Id int} \to \textbf{Id bool}$$

$$\textbf{Id}\ (\textbf{int} \to \textbf{bool}) \qquad \textbf{Id}\ (\textbf{Id}\ (\textbf{int} \to \textbf{bool})) \qquad \ldots$$

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

$$\boxed{\Delta; \Gamma \vdash e : \tau}$$

$$\ldots \qquad \frac{\Delta; \Gamma \vdash e : \tau \qquad \tau \equiv \tau'}{\Delta; \Gamma \vdash e : \tau'} \qquad \ldots$$

Type-level Expressions

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

$$\textbf{Id} \;=\; \lambda\alpha.\,\alpha$$

int $\rightarrow$ **bool**   **int** $\rightarrow$ **Id bool**   **Id int** $\rightarrow$ **bool**   **Id int** $\rightarrow$ **Id bool**

Id (int $\rightarrow$ **bool)**   **Id (Id (int** $\rightarrow$ **bool))**   ...

Admits "wrong/bad/meaningless" types:

...   **bool int**   **(Id bool) int**   **bool (Id int)**   ...

Type-level Expressions

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

$$\mathbf{Id} \;=\; \lambda\alpha.\,\alpha$$

$$\mathbf{int} \to \mathbf{bool} \qquad \mathbf{int} \to \mathbf{Id}\ \mathbf{bool} \qquad \mathbf{Id}\ \mathbf{int} \to \mathbf{bool} \qquad \mathbf{Id}\ \mathbf{int} \to \mathbf{Id}\ \mathbf{bool}$$

$$\mathbf{Id}\ (\mathbf{int} \to \mathbf{bool}) \qquad \mathbf{Id}\ (\mathbf{Id}\ (\mathbf{int} \to \mathbf{bool})) \qquad \ldots$$

Require a "type system" for types:

$$\boxed{\Delta \vdash \tau :: \kappa}$$

$$\ldots \qquad \frac{\Delta \vdash \tau_f :: \kappa_a \Rightarrow \kappa_r \qquad \Delta \vdash \tau_a :: \kappa_a}{\Delta \vdash \tau_f\ \tau_a :: \kappa_r} \qquad \ldots$$

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

- 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)

Types: $\tau ::= \mathbf{int} \mid \tau \to \tau \mid \alpha \mid \forall\alpha{::}\kappa.\ \tau \mid \lambda\alpha{::}\kappa.\ \tau \mid \tau\ \tau$

- atomic types (e.g., $\mathbf{int}$) classify the terms that evaluate to atomic values
- compound types (e.g., $\tau * \tau$) classify the terms that evaluate to compound values
- function types $\tau \to \tau$ 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: $\tau \quad ::= \quad \mathbf{int} \mid \tau \to \tau \mid \alpha \mid \forall\alpha{::}\kappa.\ \tau \mid \lambda\alpha{::}\kappa.\ \tau \mid \tau\ \tau$

- 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 $\tau \to \tau$ 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: $\tau \ ::= \ \textbf{int} \mid \tau \rightarrow \tau \mid \alpha \mid \forall\alpha{::}\kappa.\ \tau \mid \lambda\alpha{::}\kappa.\ \tau \mid \tau\ \tau$

- 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 $\kappa \ ::= \ \star \mid \kappa \Rightarrow \kappa$

- 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

Kind Examples

- $\star$
  - the kind of proper types
  - **Bool**, **Bool** $\rightarrow$ **Bool**, ...

Kind Examples

- $\star$
  - the kind of proper types
  - **Bool**, **Bool** $\rightarrow$ **Bool**, ...
- $\star \Rightarrow \star$
  - the kind of (unary) type operators
  - **List**, **Maybe**, ...

Kind Examples

- $\star$
  - the kind of proper types
  - **Bool**, **Bool** $\rightarrow$ **Bool**, **Maybe Bool**, **Maybe Bool** $\rightarrow$ **Maybe Bool**, ...
- $\star \Rightarrow \star$
  - the kind of (unary) type operators
  - **List**, **Maybe**, ...

Kind Examples

- $\star$
  - the kind of proper types
  - **Bool**, **Bool** $\rightarrow$ **Bool**, **Maybe Bool**, **Maybe Bool** $\rightarrow$ **Maybe Bool**, ...
- $\star \Rightarrow \star$
  - the kind of (unary) type operators
  - **List**, **Maybe**, ...
- $\star \Rightarrow \star \Rightarrow \star$
  - the kind of (binary) type operators
  - **Either**, **Map**, ...

Kind Examples

- $\star$
  - the kind of proper types
  - **Bool**, **Bool** $\to$ **Bool**, **Maybe Bool**, **Maybe Bool** $\to$ **Maybe Bool**, ...
- $\star \Rightarrow \star$
  - the kind of (unary) type operators
  - **List**, **Maybe**, **Map Int**, **Either** (**List Bool**), ...
- $\star \Rightarrow \star \Rightarrow \star$
  - the kind of (binary) type operators
  - **Either**, **Map**, ...

Kind Examples

- $\star$
  - the kind of proper types
  - **Bool**, **Bool** $\rightarrow$ **Bool**, **Maybe Bool**, **Maybe Bool** $\rightarrow$ **Maybe Bool**, ...
- $\star \Rightarrow \star$
  - the kind of (unary) type operators
  - **List**, **Maybe**, **Map Int**, **Either** (**List Bool**), ...
- $\star \Rightarrow \star \Rightarrow \star$
  - the kind of (binary) type operators
  - **Either**, **Map**, ...
- $(\star \Rightarrow \star) \Rightarrow \star$
  - the kind of higher-order type operators taking unary type operators to proper types
  - ???, ...

Kind Examples

- $\star$
  - the kind of proper types
  - **Bool**, **Bool** $\rightarrow$ **Bool**, **Maybe Bool**, **Maybe Bool** $\rightarrow$ **Maybe Bool**, ...
- $\star \Rightarrow \star$
  - the kind of (unary) type operators
  - **List**, **Maybe**, **Map Int**, **Either** (**List Bool**), ...
- $\star \Rightarrow \star \Rightarrow \star$
  - the kind of (binary) type operators
  - **Either**, **Map**, ...
- $(\star \Rightarrow \star) \Rightarrow \star$
  - the kind of higher-order type operators taking unary type operators to proper types
  - ???, ...
- $(\star \Rightarrow \star) \Rightarrow \star \Rightarrow \star$
  - the kind of higher-order type operators taking unary type operators to unary type operators
  - **MaybeT**, **ListT**, ...

Kind Examples

- $\star$
  - the kind of proper types
  - **Bool**, **Bool** $\rightarrow$ **Bool**, **Maybe Bool**, **Maybe Bool** $\rightarrow$ **Maybe Bool**, ...
- $\star \Rightarrow \star$
  - the kind of (unary) type operators
  - **List**, **Maybe**, **Map Int**, **Either** (**List Bool**), **ListT Maybe**, ...
- $\star \Rightarrow \star \Rightarrow \star$
  - the kind of (binary) type operators
  - **Either**, **Map**, ...
- $(\star \Rightarrow \star) \Rightarrow \star$
  - the kind of higher-order type operators taking unary type operators to proper types
  - ???, ...
- $(\star \Rightarrow \star) \Rightarrow \star \Rightarrow \star$
  - the kind of higher-order type operators taking unary type operators to unary type operators
  - **MaybeT**, **ListT**, ...

System F_ω: Syntax

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid \lambda x{:}\tau.\ e \mid e\ e \mid \Lambda\alpha{::}\kappa.\ e \mid e\ [\tau] \\
v & ::= & c \mid \lambda x{:}\tau.\ e \mid \Lambda\alpha{::}\kappa.\ e \\
\Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\tau & ::= & \mathsf{int} \mid \tau \to \tau \mid \alpha \mid \forall\alpha{::}\kappa.\ \tau \mid \lambda\alpha{::}\kappa.\ \tau \mid \tau\ \tau \\
\Delta & ::= & \cdot \mid \Delta, \alpha{::}\kappa \\
\kappa & ::= & \star \mid \kappa \Rightarrow \kappa
\end{array}
$$

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_ω: Operational Semantics

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

$$\boxed{e \rightarrow_{\text{cbv}} e'}$$

$$\frac{}{(\lambda x{:}\,\tau.\ e_b)\ v_a \rightarrow_{\text{cbv}} e_b[v_a/x]} \qquad \frac{e_f \rightarrow_{\text{cbv}} e'_f}{e_f\ e_a \rightarrow_{\text{cbv}} e'_f\ e_a}$$

$$\frac{e_a \rightarrow_{\text{cbv}} e'_a}{v_f\ e_a \rightarrow_{\text{cbv}} v_f\ e'_a} \qquad \frac{}{(\Lambda\alpha{::}\kappa_a.\ e_b)\ [\tau_a] \rightarrow_{\text{cbv}} e_b[\tau_a/\alpha]}$$

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

- *Unchanged!* All of the new action is at the type-level.

System F_ω: Type System, part 1

In the context Δ the type τ has kind κ:

$\boxed{\Delta \vdash \tau :: \kappa}$

$$\frac{}{\Delta \vdash \textbf{int} :: \star} \qquad \frac{\Delta \vdash \tau_a :: \star \qquad \Delta \vdash \tau_r :: \star}{\Delta \vdash \tau_a \to \tau_r :: \star}$$

$$\frac{\Delta(\alpha) = \kappa}{\Delta \vdash \alpha :: \kappa} \qquad \frac{\Delta, \alpha :: \kappa_a \vdash \tau_r :: \star}{\Delta \vdash \forall \alpha{::}\kappa_a.\ \tau_r :: \star}$$

$$\frac{\Delta, \alpha :: \kappa_a \vdash \tau_b :: \kappa_r}{\Delta \vdash \lambda \alpha{::}\kappa_a.\ \tau_b :: \kappa_a \Rightarrow \kappa_r} \qquad \frac{\Delta \vdash \tau_f :: \kappa_a \Rightarrow \kappa_r \qquad \Delta \vdash \tau_a :: \kappa_a}{\Delta \vdash \tau_f\ \tau_a :: \kappa_r}$$

Should look familiar:

System F_ω: Type System, part 1

In the context Δ the type τ has kind κ:

$\boxed{\Delta \vdash \tau :: \kappa}$

$$\frac{}{\Delta \vdash \textbf{int} :: \star} \qquad \frac{\Delta \vdash \tau_a :: \star \qquad \Delta \vdash \tau_r :: \star}{\Delta \vdash \tau_a \to \tau_r :: \star}$$

$$\frac{\Delta(\alpha) = \kappa}{\Delta \vdash \alpha :: \kappa} \qquad \frac{\Delta, \alpha :: \kappa_a \vdash \tau_r :: \star}{\Delta \vdash \forall \alpha{::}\kappa_a.\ \tau_r :: \star}$$

$$\frac{\Delta, \alpha :: \kappa_a \vdash \tau_b :: \kappa_r}{\Delta \vdash \lambda \alpha{::}\kappa_a.\ \tau_b :: \kappa_a \Rightarrow \kappa_r} \qquad \frac{\Delta \vdash \tau_f :: \kappa_a \Rightarrow \kappa_r \qquad \Delta \vdash \tau_a :: \kappa_a}{\Delta \vdash \tau_f\ \tau_a :: \kappa_r}$$

Should look familiar:
the typing rules of the Simply-Typed Lambda Calculus "one level up"

System F_ω: Type System, part 2

Definitional Equivalence of τ and τ':

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

$$\frac{}{\tau \equiv \tau} \qquad \frac{\tau_2 \equiv \tau_1}{\tau_1 \equiv \tau_2} \qquad \frac{\tau_1 \equiv \tau_2 \qquad \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3}$$

$$\frac{\tau_{a1} \equiv \tau_{a2} \qquad \tau_{r1} \equiv \tau_{r2}}{\tau_{a1} \to \tau_{r1} \equiv \tau_{a2} \to \tau_{r2}} \qquad \frac{\tau_{r1} \equiv \tau_{r2}}{\forall \alpha{::}\kappa_a.\ \tau_{r1} \equiv \forall \alpha{::}\kappa_a.\ \tau_{r2}}$$

$$\frac{\tau_{b1} \equiv \tau_{b2}}{\lambda \alpha{::}\kappa_a.\ \tau_{b1} \equiv \lambda \alpha{::}\kappa_a.\ \tau_{b2}} \qquad \frac{\tau_{f1} \equiv \tau_{f2} \qquad \tau_{a1} \equiv \tau_{a2}}{\tau_{f1}\ \tau_{a1} \equiv \tau_{f2}\ \tau_{a2}}$$

$$\frac{}{(\lambda \alpha{::}\kappa_a.\ \tau_b)\ \tau_a \equiv \tau_b[\alpha/\tau_a]}$$

Should look familiar:

System F$_\omega$: Type System, part 2

Definitional Equivalence of τ and τ':

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

$$\frac{}{\tau \equiv \tau} \qquad \frac{\tau_2 \equiv \tau_1}{\tau_1 \equiv \tau_2} \qquad \frac{\tau_1 \equiv \tau_2 \qquad \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3}$$

$$\frac{\tau_{a1} \equiv \tau_{a2} \qquad \tau_{r1} \equiv \tau_{r2}}{\tau_{a1} \to \tau_{r1} \equiv \tau_{a2} \to \tau_{r2}} \qquad \frac{\tau_{r1} \equiv \tau_{r2}}{\forall \alpha{::}\kappa_a.\ \tau_{r1} \equiv \forall \alpha{::}\kappa_a.\ \tau_{r2}}$$

$$\frac{\tau_{b1} \equiv \tau_{b2}}{\lambda \alpha{::}\kappa_a.\ \tau_{b1} \equiv \lambda \alpha{::}\kappa_a.\ \tau_{b2}} \qquad \frac{\tau_{f1} \equiv \tau_{f2} \qquad \tau_{a1} \equiv \tau_{a2}}{\tau_{f1}\ \tau_{a1} \equiv \tau_{f2}\ \tau_{a2}}$$

$$\frac{}{(\lambda \alpha{::}\kappa_a.\ \tau_b)\ \tau_a \equiv \tau_b[\alpha/\tau_a]}$$

Should look familiar:
the full reduction rules of the Lambda Calculus “one level up”

System F$_\omega$: Type System, part 3

In the contexts Δ and Γ the expression e has type τ:

$$\boxed{\Delta; \Gamma \vdash e : \tau}$$

$$\frac{}{\Delta; \Gamma \vdash c : \mathsf{int}} \qquad \frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau}$$

$$\frac{\Delta \vdash \tau_a :: \star \qquad \Delta; \Gamma, x : \tau_a \vdash e_b : \tau_r}{\Delta; \Gamma \vdash \lambda x{:}\tau_a.\ e_b : \tau_a \to \tau_r} \qquad \frac{\Delta; \Gamma \vdash e_f : \tau_a \to \tau_r \qquad \Delta; \Gamma \vdash e_a : \tau_a}{\Delta; \Gamma \vdash e_f\ e_a : \tau_r}$$

$$\frac{\Delta, \alpha :: \kappa_a; \Gamma \vdash e_b : \tau_r}{\Delta; \Gamma \vdash \Lambda\alpha.\ e_b : \forall\alpha{::}\kappa_a.\ \tau_r} \qquad \frac{\Delta; \Gamma \vdash e_f : \forall\alpha{::}\kappa_a.\ \tau_r \qquad \Delta \vdash \tau_a :: \kappa_a}{\Delta; \Gamma \vdash e_f\ [\tau_a] : \tau_r[\tau_a/\alpha]}$$

$$\frac{\Delta; \Gamma \vdash e : \tau \qquad \tau \equiv \tau' \qquad \Delta \vdash \tau' :: \star}{\Delta; \Gamma \vdash e : \tau'}$$

System F_ω: Type System, part 3

In the contexts Δ and Γ the expression e has type τ:

$\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\frac{}{\Delta; \Gamma \vdash c : \mathsf{int}} \qquad \frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau}$$

$$\frac{\Delta \vdash \tau_a :: \star \qquad \Delta; \Gamma, x : \tau_a \vdash e_b : \tau_r}{\Delta; \Gamma \vdash \lambda x{:}\tau_a.\ e_b : \tau_a \to \tau_r} \qquad \frac{\Delta; \Gamma \vdash e_f : \tau_a \to \tau_r \qquad \Delta; \Gamma \vdash e_a : \tau_a}{\Delta; \Gamma \vdash e_f\ e_a : \tau_r}$$

$$\frac{\Delta, \alpha :: \kappa_a; \Gamma \vdash e_b : \tau_r}{\Delta; \Gamma \vdash \Lambda\alpha.\ e_b : \forall\alpha{::}\kappa_a.\ \tau_r} \qquad \frac{\Delta; \Gamma \vdash e_f : \forall\alpha{::}\kappa_a.\ \tau_r \qquad \Delta \vdash \tau_a :: \kappa_a}{\Delta; \Gamma \vdash e_f\ [\tau_a] : \tau_r[\tau_a/\alpha]}$$

$$\frac{\Delta; \Gamma \vdash e : \tau \qquad \tau \equiv \tau' \qquad \Delta \vdash \tau' :: \star}{\Delta; \Gamma \vdash e : \tau'}$$

Syntax and type system easily extended with recursive and existential types.

Polymorphic List Library with higher-order ∃

List library is an existential package:

$$\begin{array}{l}
\texttt{pack}(\boldsymbol{\lambda\alpha{::}\star.\ \mu\xi{::}\star.}\ \textbf{unit} + (\alpha * \xi), \mathit{list_library}) \\
\texttt{as}\ \exists L{::}\star \Rightarrow \star.\ \{\textbf{empty} : \forall\alpha{::}\star.\ L\ \alpha; \\
\qquad\qquad\qquad\quad \textbf{cons} : \forall\alpha{::}\star.\ \alpha \to L\ \alpha \to L\ \alpha; \\
\qquad\qquad\qquad\quad \textbf{unlist} : \forall\alpha{::}\star.\ L\ \alpha \to \textbf{unit} + (\alpha * L\ \alpha); \\
\qquad\qquad\qquad\quad \textbf{map} : \forall\alpha{::}\star.\ \forall\beta{::}\star.\ (\alpha \to \beta) \to L\ \alpha \to L\ \beta; \\
\qquad\qquad\qquad\quad \ldots\}
\end{array}$$

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

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

List operations are polymorphic in element type.

The **map** function only allows mapping α lists to β 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_ω is type safe.

Metatheory

System F_ω 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 $\Delta_1, \Delta_2; \Gamma_1, \Gamma_2[\tau_2/\alpha] \vdash e_1[\tau_2/\alpha] : \tau$.

  - All straightforward inductions, using various weakening and exchange lemmas.

Metatheory

System F_ω is type safe.

- Progress:
  Induction on typing derivation, using canonical form lemmas:
  - If $\cdot; \cdot \vdash v : \mathbf{int}$, then $v = c$.
  - If $\cdot; \cdot \vdash v : \tau_a \to \tau_r$, 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:

$$\frac{\Delta; \Gamma \vdash e : \tau \qquad \tau \equiv \tau' \qquad \Delta \vdash \tau' :: \star}{\Delta; \Gamma \vdash e : \tau'}$$

  (just like with subtyping and subsumption).

Definitional Equivalence and Parallel Reduction

Parallel Reduction of τ to τ':

$\boxed{\tau \Rrightarrow \tau'}$

$$\frac{}{\tau \Rrightarrow \tau}$$

$$\frac{\tau_{a1} \Rrightarrow \tau_{a2} \qquad \tau_{r1} \Rrightarrow \tau_{r2}}{\tau_{a1} \to \tau_{r1} \Rrightarrow \tau_{a2} \to \tau_{r2}} \qquad \frac{\tau_{r1} \Rrightarrow \tau_{r2}}{\forall\alpha{::}\kappa_a.\ \tau_{r1} \Rrightarrow \forall\alpha{::}\kappa_a.\ \tau_{r2}}$$

$$\frac{\tau_{b1} \Rrightarrow \tau_{b2}}{\lambda\alpha{::}\kappa_a.\ \tau_{b1} \Rrightarrow \lambda\alpha{::}\kappa_a.\ \tau_{b2}} \qquad \frac{\tau_{f1} \Rrightarrow \tau_{f2} \qquad \tau_{a1} \Rrightarrow \tau_{a2}}{\tau_{f1}\ \tau_{a1} \Rrightarrow \tau_{f2}\ \tau_{a2}}$$

$$\frac{\tau_b \Rrightarrow \tau_b' \qquad \tau_a \Rrightarrow \tau_a'}{(\lambda\alpha{::}\kappa_a.\ \tau_b)\ \tau_a \Rrightarrow \tau_b'[\alpha/\tau_a']}$$

A more “computational” relation.

Definitional Equivalence and Parallel Reduction

Key properties:

Definitional Equivalence and Parallel Reduction

Key properties:

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

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:
  - If $\tau \Rrightarrow^* \tau_1$ and $\tau \Rrightarrow^* \tau_2$,
    then there exists τ' such that $\tau_1 \Rrightarrow^* \tau'$ and $\tau_2 \Rrightarrow^* \tau'$

Definitional Equivalence and Parallel Reduction

Key properties:

- Transitive and symmetric closure of parallel reduction and type equivalence coincide:
  - $\boldsymbol{\tau} \Leftrightarrow^* \boldsymbol{\tau'}$ iff $\boldsymbol{\tau} \equiv \boldsymbol{\tau'}$
- Parallel reduction has the Church-Rosser property:
  - If $\boldsymbol{\tau} \Rrightarrow^* \boldsymbol{\tau_1}$ and $\boldsymbol{\tau} \Rrightarrow^* \boldsymbol{\tau_2}$,
    then there exists $\boldsymbol{\tau'}$ such that $\boldsymbol{\tau_1} \Rrightarrow^* \boldsymbol{\tau'}$ and $\boldsymbol{\tau_2} \Rrightarrow^* \boldsymbol{\tau'}$
- Equivalent types share a common reduct:
  - If $\boldsymbol{\tau_1} \equiv \boldsymbol{\tau_2}$, then there exists $\boldsymbol{\tau'}$ such that $\boldsymbol{\tau_1} \Rrightarrow^* \boldsymbol{\tau'}$ and $\boldsymbol{\tau_2} \Rrightarrow^* \boldsymbol{\tau'}$

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:
  - If $\tau \Rrightarrow^* \tau_1$ and $\tau \Rrightarrow^* \tau_2$,
    then there exists τ' such that $\tau_1 \Rrightarrow^* \tau'$ and $\tau_2 \Rrightarrow^* \tau'$
- Equivalent types share a common reduct:
  - If $\tau_1 \equiv \tau_2$, then there exists τ' such that $\tau_1 \Rrightarrow^* \tau'$ and $\tau_2 \Rrightarrow^* \tau'$
- Reduction preserves shapes:
  - If **int** $\Rrightarrow^* \tau'$, then $\tau' =$ **int**
  - If $\tau_a \rightarrow \tau_r \Rrightarrow^* \tau'$, then $\tau' = \tau'_a \rightarrow \tau'_r$ and $\tau_a \Rrightarrow^* \tau'_a$ and $\tau_r \Rrightarrow^* \tau'_r$
  - If $\forall\alpha{::}\kappa_a.\ \tau_r \Rrightarrow^* \tau'$, then $\tau' = \forall\alpha{::}\kappa_a.\ \tau'_r$ and $\tau_r \Rrightarrow^* \tau'_r$

Canonical Forms

If $\cdot;\cdot \vdash v : \tau_a \to \tau_r$, then $v = \lambda x{:}\tau_a.\ e_b$.
Proof:
By cases on the form of v:

Canonical Forms

If $\cdot;\cdot \vdash v : \tau_a \to \tau_r$, 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 $\cdot;\cdot \vdash v : \tau_a \to \tau_r$, then $v = \lambda x{:}\tau_a.\ e_b$.
Proof:
By cases on the form of v:

- $v = c$.
  Derivation of $\cdot;\cdot \vdash v : \tau_a \to \tau_r$ must be of the form:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\cdot;\cdot \vdash c : \mathbf{int}} \quad \mathbf{int} \equiv \tau_1}{\cdot;\cdot \vdash c : \tau_1}}{\begin{array}{c}\vdots \\ \cdot;\cdot \vdash c : \tau_{n-1}\end{array} \quad \tau_{n-1} \equiv \tau_n}}{\cdot;\cdot \vdash c : \tau_n} \quad \tau_n \equiv \tau_a \to \tau_r$$
$$\overline{\cdot;\cdot \vdash c : \tau_a \to \tau_r}$$

  Therefore, we can construct the derivation $\mathbf{int} \equiv \tau_a \to \tau_r$.
  We can find a common reduct: $\mathbf{int} \Rrightarrow^* \tau^\dagger$ and $\tau_a \to \tau_r \Rrightarrow^* \tau^\dagger$.
  Reduction preserves shape: $\mathbf{int} \Rrightarrow^* \tau^\dagger$ implies $\tau^\dagger = \mathbf{int}$.
  Reduction preserves shape: $\tau_a \to \tau_r \Rrightarrow^* \tau^\dagger$ implies $\tau^\dagger = \tau'_a \to \tau'_r$.
  But, $\tau^\dagger = \mathbf{int}$ and $\tau^\dagger = \tau'_a \to \tau'_r$ is a contradiction.

Canonical Forms

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

Proof:

By cases on the form of v:

- $v = \Lambda\alpha{::}\kappa_a.\ e_b$.
  Derivation of $\cdot;\cdot \vdash v : \tau_a \to \tau_r$ must be of the form:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\cdot;\cdot \vdash \Lambda\alpha{::}\kappa_a.\ e_b : \forall\alpha{::}\kappa_a.\ \tau_z} \quad \forall\alpha{::}\kappa_a.\ \tau_z \equiv \tau_1}{\cdot;\cdot \vdash \Lambda\alpha{::}\kappa_a.\ e_b : \tau_1}}{\begin{array}{c}\vdots\\ \cdot;\cdot \vdash \Lambda\alpha{::}\kappa_a.\ e_b : \tau_{n-1}\end{array} \quad \tau_{n-1} \equiv \tau_n}}{\cdot;\cdot \vdash \Lambda\alpha{::}\kappa_a.\ e_b : \tau_n} \quad \tau_n \equiv \tau_a \to \tau_r}{\cdot;\cdot \vdash \Lambda\alpha{::}\kappa_a.\ e_b : \tau_a \to \tau_r}$$

  Therefore, we can construct the derivation $\forall\alpha{::}\kappa_a.\ \tau_z \equiv \tau_a \to \tau_r$.
  We can find a common reduct: $\forall\alpha{::}\kappa_a.\ \tau_z \Rrightarrow^* \tau^\dagger$ and $\tau_a \to \tau_r \Rrightarrow^* \tau^\dagger$.
  Reduction preserves shape: $\forall\alpha{::}\kappa_a.\ \tau_z \Rrightarrow^* \tau^\dagger$ implies $\tau^\dagger = \forall\alpha{::}\kappa_a.\ \tau_z'$.
  Reduction preserves shape: $\tau_a \to \tau_r \Rrightarrow^* \tau^\dagger$ implies $\tau^\dagger = \tau_a' \to \tau_r'$.
  But, $\tau^\dagger = \forall\alpha{::}\kappa_a.\ \tau_z'$ and $\tau^\dagger = \tau_a' \to \tau_r'$ is a contradiction.

Metatheory

System F_ω is type safe.

Where was the $\boldsymbol{\Delta} \vdash \boldsymbol{\tau} :: \boldsymbol{\kappa}$ judgement used in the proof?

Metatheory

System F_ω is type safe.

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

Metatheory

System F_{ω} is type safe.

Where was the $\boldsymbol{\Delta} \vdash \boldsymbol{\tau} :: \boldsymbol{\kappa}$ 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_ω without Kinds / System F with Type-Level Abstraction and Application

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda x{:}\tau.\ e \mid e\ e \mid \Lambda\alpha.\ e \mid e\ [\tau] \\
v & ::= & c \mid \lambda x{:}\tau.\ e \mid \Lambda\alpha.\ e \\
\tau & ::= & \mathsf{int} \mid \tau \to \tau \mid \alpha \mid \forall\alpha.\ \tau \mid \lambda\alpha.\ \tau \mid \tau\ \tau
\end{array}
$$

$$
\begin{array}{lcl}
\Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\Delta & ::= & \cdot \mid \Delta, \alpha
\end{array}
$$

System F_ω without Kinds / System F with Type-Level Abstraction and Application

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda x{:}\tau.\ e \mid e\ e \mid \Lambda\alpha.\ e \mid e\ [\tau] \\
v & ::= & c \mid \lambda x{:}\tau.\ e \mid \Lambda\alpha.\ e \\
\tau & ::= & \mathsf{int} \mid \tau \rightarrow \tau \mid \alpha \mid \forall\alpha.\ \tau \mid \lambda\alpha.\ \tau \mid \tau\ \tau
\end{array}
$$

$$
\begin{array}{lcl}
\Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\Delta & ::= & \cdot \mid \Delta, \alpha
\end{array}
$$

$\boxed{e \rightarrow_{\mathsf{cbv}} e'}$

$$
\frac{}{(\lambda x{:}\ \tau.\ e_b)\ v_a \rightarrow_{\mathsf{cbv}} e_b[v_a/x]}
\qquad
\frac{e_f \rightarrow_{\mathsf{cbv}} e_f'}{e_f\ e_a \rightarrow_{\mathsf{cbv}} e_f'\ e_a}
\qquad
\frac{e_a \rightarrow_{\mathsf{cbv}} e_a'}{v_f\ e_a \rightarrow_{\mathsf{cbv}} v_f\ e_a'}
$$

$$
\frac{}{(\Lambda\alpha.\ e_b)\ [\tau_a] \rightarrow_{\mathsf{cbv}} e_b[\tau_a/\alpha]}
\qquad
\frac{e_f \rightarrow_{\mathsf{cbv}} e_f'}{e_f\ [\tau_a] \rightarrow_{\mathsf{cbv}} e_f'\ [\tau_a]}
$$

System F_ω without Kinds / System F with Type-Level Abstraction and Application

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda x{:}\tau.\ e \mid e\ e \mid \Lambda\alpha.\ e \mid e\ [\tau] \\
v & ::= & c \mid \lambda x{:}\tau.\ e \mid \Lambda\alpha.\ e \\
\tau & ::= & \mathsf{int} \mid \tau \to \tau \mid \alpha \mid \forall\alpha.\ \tau \mid \lambda\alpha.\ \tau \mid \tau\ \tau
\end{array}
\qquad
\begin{array}{lcl}
\Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\Delta & ::= & \cdot \mid \Delta, \alpha
\end{array}
$$

$\boxed{\Delta \vdash \tau :: \checkmark}$

$$
\frac{}{\Delta \vdash \mathsf{int} :: \checkmark}
\qquad
\frac{\Delta \vdash \tau_a :: \checkmark \qquad \Delta \vdash \tau_r :: \checkmark}{\Delta \vdash \tau_a \to \tau_r :: \checkmark}
$$

$$
\frac{\alpha \in \Delta}{\Delta \vdash \alpha :: \checkmark}
\qquad
\frac{\Delta, \alpha \vdash \tau_r :: \checkmark}{\Delta \vdash \forall\alpha.\ \tau_r :: \checkmark}
$$

$$
\frac{\Delta, \alpha \vdash \tau_b :: \checkmark}{\Delta \vdash \lambda\alpha.\ \tau_b :: \checkmark}
\qquad
\frac{\Delta \vdash \tau_f :: \checkmark \qquad \Delta \vdash \tau_a :: \checkmark}{\Delta \vdash \tau_f\ \tau_a :: \checkmark}
$$

Check that free type variables of τ are in Δ, but nothing else.

System F_ω without Kinds / System F with Type-Level Abstraction and Application

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda x{:}\tau.\ e \mid e\ e \mid \Lambda\alpha.\ e \mid e\ [\tau] \\
v & ::= & c \mid \lambda x{:}\tau.\ e \mid \Lambda\alpha.\ e \\
\tau & ::= & \mathsf{int} \mid \tau \to \tau \mid \alpha \mid \forall\alpha.\ \tau \mid \lambda\alpha.\ \tau \mid \tau\ \tau
\end{array}
$$

$$
\begin{array}{lcl}
\Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\Delta & ::= & \cdot \mid \Delta, \alpha
\end{array}
$$

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

$$
\frac{}{\tau \equiv \tau}
\qquad
\frac{\tau_2 \equiv \tau_1}{\tau_1 \equiv \tau_2}
\qquad
\frac{\tau_1 \equiv \tau_2 \qquad \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3}
$$

$$
\frac{\tau_{a1} \equiv \tau_{a2} \qquad \tau_{r1} \equiv \tau_{r2}}{\tau_{a1} \to \tau_{r1} \equiv \tau_{a2} \to \tau_{r2}}
\qquad
\frac{\tau_{r1} \equiv \tau_{r2}}{\forall\alpha.\ \tau_{r1} \equiv \forall\alpha.\ \tau_{r2}}
$$

$$
\frac{\tau_{b1} \equiv \tau_{b2}}{\lambda\alpha.\ \tau_{b1} \equiv \lambda\alpha.\ \tau_{b2}}
\qquad
\frac{\tau_{f1} \equiv \tau_{f2} \qquad \tau_{a1} \equiv \tau_{a2}}{\tau_{f1}\ \tau_{a1} \equiv \tau_{f2}\ \tau_{a2}}
$$

$$
\frac{}{(\lambda\alpha.\ \tau_b)\ \tau_a \equiv \tau_b[\alpha/\tau_a]}
$$

System F_ω without Kinds / System F with Type-Level Abstraction and Application

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda x{:}\tau.\ e \mid e\ e \mid \Lambda\alpha.\ e \mid e\ [\tau] \\
v & ::= & c \mid \lambda x{:}\tau.\ e \mid \Lambda\alpha.\ e \\
\tau & ::= & \mathsf{int} \mid \tau \to \tau \mid \alpha \mid \forall\alpha.\ \tau \mid \lambda\alpha.\ \tau \mid \tau\ \tau
\end{array}
$$

$$
\begin{array}{lcl}
\Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\Delta & ::= & \cdot \mid \Delta, \alpha
\end{array}
$$

$\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\frac{}{\Delta; \Gamma \vdash c : \mathsf{int}} \qquad \frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau}$$

$$\frac{\Delta \vdash \tau_a :: \checkmark \qquad \Delta; \Gamma, x : \tau_a \vdash e_b : \tau_r}{\Delta; \Gamma \vdash \lambda x{:}\tau_a.\ e_b : \tau_a \to \tau_r} \qquad \frac{\Delta; \Gamma \vdash e_f : \tau_a \to \tau_r \qquad \Delta; \Gamma \vdash e_a : \tau_a}{\Delta; \Gamma \vdash e_f\ e_a : \tau_r}$$

$$\frac{\Delta, \alpha; \Gamma \vdash e_b : \tau_r}{\Delta; \Gamma \vdash \Lambda\alpha.\ e_b : \forall\alpha.\ \tau_r} \qquad \frac{\Delta; \Gamma \vdash e_f : \forall\alpha.\ \tau_r \qquad \Delta \vdash \tau_a :: \checkmark}{\Delta; \Gamma \vdash e_f\ [\tau_a] : \tau_r[\tau_a/\alpha]}$$

$$\frac{\Delta; \Gamma \vdash e : \tau \qquad \tau \equiv \tau'}{\Delta; \Gamma \vdash e : \tau'}$$

This language is type safe.

This language is type safe.

- Preservation:
  Induction on typing derivation, using substitution lemmas:
  - Term Substitution:
    if $\boldsymbol{\Delta_1, \Delta_2; \Gamma_1, x : \tau_x, \Gamma_2 \vdash e_1 : \tau}$ and $\boldsymbol{\Delta_1; \Gamma_1 \vdash e_2 : \tau_x}$,
    then $\boldsymbol{\Delta_1, \Delta_2; \Gamma_1, \Gamma_2 \vdash e_1[e_2/x] : \tau}$.
  - Type Substitution:
    if $\boldsymbol{\Delta_1, \alpha, \Delta_2 \vdash \tau_1 :: \checkmark}$ and $\boldsymbol{\Delta_1 \vdash \tau_2 :: \checkmark}$,
    then $\boldsymbol{\Delta_1, \Delta_2 \vdash \tau_1[\tau_2/\alpha] :: \checkmark}$.
  - Type Substitution:
    if $\boldsymbol{\tau_1 \equiv \tau_2}$, then $\boldsymbol{\tau_1[\tau/\alpha] \equiv \tau_2[\tau/\alpha]}$.
  - Type Substitution:
    if $\boldsymbol{\Delta_1, \alpha, \Delta_2; \Gamma_1, \Gamma_2 \vdash e_1 : \tau}$ and $\boldsymbol{\Delta_1 \vdash \tau_2 :: \checkmark}$,
    then $\boldsymbol{\Delta_1, \Delta_2; \Gamma_1, \Gamma_2[\tau_2/\alpha] \vdash e_1[\tau_2/\alpha] : \tau}$.

  - All straightforward inductions, using various weakening and exchange lemmas.

This language is type safe.

- Progress:
  Induction on typing derivation, using canonical form lemmas:
  - If $\cdot;\cdot \vdash v : \mathbf{int}$, then $v = c$.
  - If $\cdot;\cdot \vdash v : \tau_a \to \tau_r$, 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:

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

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 τ and τ' are the same.

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

$$\dfrac{\dfrac{\dfrac{\vdots}{\Delta;\Gamma \vdash e_f : \tau_f} \quad \dfrac{\vdots}{\tau_f \equiv \tau_a \to \tau_r}}{\Delta;\Gamma \vdash e_f : \tau_a \to \tau_r} \quad \dfrac{\vdots}{\Delta;\Gamma \vdash e_a : \tau_a}}{\Delta;\Gamma \vdash e_f\ e_a : \tau_r}$$

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 τ and τ' are the same.

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

- E.g., $(\lambda\alpha.\ \alpha \to \alpha)\ (\textbf{int int}) \equiv (\textbf{int int}) \to (\textbf{int int})$

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 τ and τ' are the same.

Definitional equivalence ($\tau \equiv \tau'$) and parallel reduction ($\tau \Rrightarrow \tau'$)
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-kinded 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-kinded 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.

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 $\mathbf{\Delta}$, $\mathbf{\Gamma}$, and e, does there exist $\boldsymbol{\tau}$ such that $\mathbf{\Delta};\mathbf{\Gamma} \vdash e : \boldsymbol{\tau}$.

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

Two issues:

- $\dfrac{\boldsymbol{\Delta};\boldsymbol{\Gamma} \vdash e : \boldsymbol{\tau} \qquad \boldsymbol{\tau} \equiv \boldsymbol{\tau}' \qquad \boldsymbol{\Delta} \vdash \boldsymbol{\tau}' :: \star}{\boldsymbol{\Delta};\boldsymbol{\Gamma} \vdash e : \boldsymbol{\tau}'}$ is a non-syntax-directed rule
- $\boldsymbol{\tau} \equiv \boldsymbol{\tau}'$ is a non-syntax-directed relation

One non-issue:

- $\boldsymbol{\Delta} \vdash \boldsymbol{\tau} :: \boldsymbol{\kappa}$ is a syntax-directed relation (STLC "one level up")

Type Checking for System F_ω

Remove non-syntax-directed rules and relations:

$\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\frac{}{\Delta; \Gamma \vdash c : \mathsf{int}} \qquad \frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau}$$

$$\frac{\Delta \vdash \tau_a :: \star \qquad \Delta; \Gamma, x : \tau_a \vdash e_b : \tau_r}{\Delta; \Gamma \vdash \lambda x{:}\tau_a.\ e_b : \tau_a \to \tau_r} \qquad \frac{\Delta, \alpha{::}\kappa_a; \Gamma \vdash e_b : \tau_r}{\Delta; \Gamma \vdash \Lambda\alpha.\ e_b : \forall\alpha{::}\kappa_a.\ \tau_r}$$

$$\frac{\begin{array}{ccc} \Delta; \Gamma \vdash e_f : \tau_f & \tau_f \Rrightarrow^{\Downarrow} \tau_f' & \tau_f' = \tau_{fa}' \to \tau_{fr}' \\ \Delta; \Gamma \vdash e_a : \tau_a & \tau_a \Rrightarrow^{\Downarrow} \tau_a' & \tau_{fa}' = \tau_a' \end{array}}{\Delta; \Gamma \vdash e_f\ e_a : \tau_{fr}'}$$

$$\frac{\begin{array}{ccc} \Delta; \Gamma \vdash e_f : \tau_f & \tau_f \Rrightarrow^{\Downarrow} \tau_f' & \tau_f' = \forall\alpha{::}\kappa_{fa}.\ \tau_{fr} \\ \Delta \vdash \tau_a :: \kappa_a & & \kappa_{fa} = \kappa_a \end{array}}{\Delta; \Gamma \vdash e_f\ [\tau_a] : \tau_{fr}[\tau_a/\alpha]}$$

Type Checking for System F_ω

Kinds are for *type checking*.

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

Metatheory for kind system:

Type Checking for System F_ω

Kinds are for *type checking*.

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

Metatheory for kind system:

- Well-kinded types don't get stuck.
  - If $\boldsymbol{\Delta} \vdash \boldsymbol{\tau} :: \boldsymbol{\kappa}$ and $\boldsymbol{\tau} \Rrightarrow^* \boldsymbol{\tau}'$, then either $\boldsymbol{\tau}'$ is in (weak-head) normal form (i.e., a type-level "value") or $\boldsymbol{\tau}' \Rrightarrow \boldsymbol{\tau}''$.
  - Proofs by Progress and Preservation on kinding and parallel reduction derivations.

Type Checking for System F_ω

Kinds are for *type checking*.

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

Metatheory for kind system:

- Well-kinded types don't get stuck.
  - If $\Delta \vdash \tau :: \kappa$ and $\tau \Rrightarrow^* \tau'$,
    then either τ' is in (weak-head) normal form (i.e., a type-level "value") or $\tau' \Rrightarrow \tau''$.
  - Proofs by Progress and Preservation on kinding and parallel reduction derivations.
  - But, irrelevant for type checking of expressions.
    If $\tau_f \Rrightarrow^* \tau'_f$ "gets stuck" at a type τ'_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_ω

Kinds are for *type checking*.

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

Metatheory for kind system:

- Well-kinded types don't get stuck.
  - If $\mathbf{\Delta} \vdash \tau :: \kappa$ and $\tau \Rrightarrow^* \tau'$, then either τ' is in (weak-head) normal form (i.e., a type-level "value") or $\tau' \Rrightarrow \tau''$.
  - 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 $\Delta; \Gamma \vdash e : \tau$.

Metatheory for kind system:

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

Type Checking for System F_ω

Kinds are for *type checking*.

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

Metatheory for kind system:

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

Type checking for System F_ω 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."