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”
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:

pack(µξ. unit + (int * ξ), list_library)
as ∃L. {empty : L;
  cons : int → L → L;
  unlist : L → unit + (int * L);
  map : (int → int) → L → L;
  ...
}

The witness type is integer lists: µξ. unit + (int * ξ).

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

Λα. pack(µξ. unit + (α * ξ), list_library)
as ∃L. {empty : L;
  cons : α → L → L;
  unlist : L → unit + (α * L);
  map : (α → α) → L → L;
  ...
}

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.

(Independent) List Library with ∀/∃

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

pack(µξ. unit + (α * ξ), list_library)
as ∃L. {empty : L;
  cons : α → L → L;
  unlist : L → unit + (α * L);
  map : (α → α) → L → L;
  ...
}

The witness type is α lists: µξ. unit + (α * ξ).

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:

L α = µξ. unit + (α * ξ)

- replace occurrences of L τ in programs
  with (µξ. unit + (α * ξ))[τ/α]

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

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.

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

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

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

\[ \tau \equiv \tau' \]

\[
\begin{align*}
\cdots & \quad (\lambda \alpha. \tau_b) \tau_a \equiv \tau_b[\alpha/\tau_a] \\
\end{align*}
\]

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

\[
\frac{\Delta; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash e : \tau'}
\]
Type-level Expressions

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

\[ \text{Id} = \lambda \alpha. \alpha \]
\[ \text{int} \to \text{bool} \quad \text{int} \to \text{Id} \to \text{bool} \quad \text{Id} \to \text{int} \to \text{Id} \to \text{bool} \quad \text{Id} \to \text{bool} \]
\[ 
\text{Id} (\text{int} \to \text{bool}) \quad \text{Id} (\text{Id} (\text{int} \to \text{bool})) \quad \ldots
\]

Admits "wrong/bad/meaningless" types:

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

Require a "type system" for types:

\[ \Delta \vdash \tau :: \kappa \]
\[ \ldots \quad \Delta \vdash \tau f :: \kappa_a \Rightarrow \kappa_r \quad \Delta \vdash \tau a :: \kappa_a \]
\[ \ldots \quad \Delta \vdash \tau f \tau a :: \kappa_r \]

Terms, Types, and Kinds, Oh My

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

- atomic values (e.g., \( c \)) and operations (e.g., \( e + e \))
- compound values (e.g., \( (e, v) \)) and operations (e.g., \( e \cdot 1 \))
- 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:  
\[ e ::= c \mid x \mid \lambda x : \tau. \ e \mid e \ e \mid \Lambda \alpha :: \kappa. \ e \mid e [\tau] \]

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

- atomic types (e.g., `int`) classify the terms that evaluate to atomic values
- compound types (e.g., `\tau \rightarrow \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)

Kind Examples

Kinds  
\[ \kappa ::= * \mid \kappa \Rightarrow \kappa \]

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

- Taking unary type operators to proper types
  - `MaybeT, ListT, ...`
System Fω: Syntax

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

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

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

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

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

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

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

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

System Fω: Type System, part 1

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

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

- \(\Delta \vdash \text{int} :: \star\)
- \(\Delta \vdash \tau_a :: \star\)
- \(\Delta \vdash \tau_r :: \star\)
- \(\Delta \vdash \alpha :: \kappa\)
- \(\Delta \vdash \forall \alpha :: \kappa_a. \tau_r :: \star\)

Should look familiar:

the typing rules of the Simply-Typed Lambda Calculus “one level up”

System Fω: Operational Semantics

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

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

\[
(\lambda x: \tau. e_b) v_a \rightarrow_{cbv} e_b[v_a/x] \quad e_f \rightarrow_{cbv} e_f'
\]

\[
e_f e_a \rightarrow_{cbv} e_f e_a\]

\[
\frac{e_a \rightarrow_{cbv} e_a'}{v_f e_a \rightarrow_{cbv} v_f e_a'} \quad (\Lambda \alpha :: \kappa_a. e_b) [\tau_a] \rightarrow_{cbv} e_b[\tau_a/\alpha]
\]

\[
\frac{e_f \rightarrow_{cbv} e_f'}{e_f [\tau_a] \rightarrow_{cbv} e_f [\tau_a]}
\]

Should look familiar:

\[
\begin{align*}
\frac{\Delta \vdash \tau_a :: \star}{\Delta \vdash \tau_r :: \star} \\
\frac{\Delta \vdash \alpha :: \kappa_a \Rightarrow \kappa_r}{\Delta \vdash \alpha :: \kappa_a \Rightarrow \kappa_r} \\
\frac{\Delta \vdash \forall \alpha :: \kappa_a. \tau_r :: \star}{\Delta \vdash \forall \alpha :: \kappa_a. \tau_r :: \star} \\
\end{align*}
\]

the typing rules of the Simply-Typed Lambda Calculus “one level up”
System Fω: Type System, part 2

Definitional Equivalence of \( \tau \) and \( \tau' \):

\[
\begin{array}{c}
\tau \equiv \tau' \\
\tau_1 \equiv \tau_2 \\
\tau_2 \equiv \tau_1 \\
\tau_1 \equiv \tau_3 \\
\tau_3 \equiv \tau_2 \\
\tau_{a_1} \equiv \tau_{a_2} \\
\tau_{r_1} \equiv \tau_{r_2} \\
\forall \alpha :: \kappa_a \cdot \tau_{r_1} \equiv \forall \alpha :: \kappa_a \cdot \tau_{r_2} \\
\lambda \alpha :: \kappa_a \cdot \tau_{b_1} \equiv \lambda \alpha :: \kappa_a \cdot \tau_{b_2} \\
\lambda \alpha :: \kappa_a \cdot \tau_f \equiv \lambda \alpha :: \kappa_a \cdot \tau_f \\
\end{array}
\]

\( (\lambda \alpha :: \kappa_a \cdot \tau_b) \tau_a \equiv \tau_b[\alpha/\tau_a] \)

Should look familiar:

System Fω: Type System, part 3

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The existential \textit{type operator} variable $L$ represents poly. lists.

List operations are polymorphic in element type.

The \text{map} function only 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.

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

System Fω is type safe.

- Progress:
  Induction on typing derivation, using canonical form lemmas:
  - If ·; · ⊢ v : int, then v = c.
  - If ·; · ⊢ v : τa → τr, then v = λx:τa. e_b.
  - If ·; · ⊢ v : ∀α::κa. τr, then v = Δα::κa. e_b.
  - Complicated by typing derivations that end with:

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

(just like with subtyping and subsumption).

Definitional Equivalence and Parallel Reduction

Key properties:
- Transitive and symmetric closure of parallel reduction and type equivalence coincide:
  - \( \tau \iff^* \tau' \) 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 \( \tau \Rightarrow^* \tau_1 \) 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:
  - \( \tau \leftrightarrow^* \tau' \iff \tau \equiv \tau' \)
- 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 \( \tau_2 \Rightarrow^* \tau' \)

Canonical Forms

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

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

System $F_\omega$ is type safe.

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

 Canonical Forms
If $\cdot; \vdash v : \tau_a \rightarrow \tau_r$, then $v = \lambda x : \tau_a . \ e_b$.
Proof:
By cases on the form of $v$:
► $v = \lambda x : \tau_a . \ e_b$.
Derivation of $\cdot; \vdash v : \tau_a \rightarrow \tau_r$ must be of the form:
\[
\frac{\cdot; \vdash c : \text{int} \quad \text{int} \equiv \tau_1}{\cdot; \vdash c : \tau_1}
\]
\[
\vdots
\]
\[
\frac{\cdot; \vdash c : \tau_{n-1} \quad \tau_{n-1} \equiv \tau_n}{\cdot; \vdash c : \tau_n}
\]
\[
\vdots
\]
\[
\frac{\cdot; \vdash c : \tau_a \rightarrow \tau_r}{\cdot; \vdash c : \tau_a \rightarrow \tau_r}
\]
Therefore, we can construct the derivation $\text{int} \equiv \tau_a \rightarrow \tau_r$.
We can find a common reduct: $\text{int} \Rightarrow^* \tau^\dagger$ and $\tau_a \rightarrow \tau_r \Rightarrow^* \tau^\dagger$.
Reduction preserves shape: $\text{int} \Rightarrow^* \tau^\dagger$ implies $\tau^\dagger = \text{int}$.
Reduction preserves shape: $\tau_a \rightarrow \tau_r \Rightarrow^* \tau^\dagger$ implies $\tau^\dagger = \tau_a' \rightarrow \tau_r'$.
But, $\tau^\dagger = \text{int}$ 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?
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!

Matthew Fluet
CSE-505 2015, Lecture 27

---

System $F_\omega$ without Kinds / System F with Type-Level Abstraction and Application

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

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

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

\[
(\lambda x: \tau. e_b) \ v_a \rightarrow_{\text{cbv}} e_b[v_a/x]  \\
\frac{ef \rightarrow_{\text{cbv}} e'_f}{ef \ e_a \rightarrow_{\text{cbv}} e'_f \ e_a}  \\
\frac{e_a \rightarrow_{\text{cbv}} e'_a}{vf \ e_a \rightarrow_{\text{cbv}} v_f \ e_a}
\]

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

Matthew Fluet
CSE-505 2015, Lecture 27

24

Matthew Fluet
CSE-505 2015, Lecture 27

25
System $F$ 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.
This language 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 \( \Delta_1, \alpha, \Delta_2 \vdash \tau_1 :: \check{\gamma} \) and \( \Delta_1 \vdash \tau_2 :: \check{\gamma} \),
  then \( \Delta_1, \Delta_2 \vdash \tau_1[\tau_2/\alpha] :: \check{\gamma} \).
- **Type Substitution:**
  if \( \tau_1 \equiv \tau_2 \), then \( \tau_1[\tau/\alpha] \equiv \tau_2[\tau/\alpha] \).
- **Type Substitution:**
  if \( \Delta_1, \alpha, \Delta_2; \Gamma_1, \Gamma_2 \vdash e_1 : \tau \) and \( \Delta_1 \vdash \tau_2 :: \check{\gamma} \),
  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.

**Progress:**
Induction on typing derivation, using canonical form lemmas:

- If \( :: \vdash v : \text{int} \), then \( v = c \).
- If \( :: \vdash v : \tau_a \rightarrow \tau_r \), then \( v = \lambda x : \tau_a . \ e_b \).
- If \( :: \vdash v : \forall \alpha. \tau_r \), then \( v = \Lambda \alpha . \ e_b \).
- Using parallel reduction relation.

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

Recall statement of type safety:

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

Why aren’t kinds required for type safety?

Recall statement of type safety:

If \( \vdash \cdot \vdash e : \tau \), then \( e \) does not get stuck.

The typing derivation \( \vdash \cdot \vdash e : \tau \)
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:

\[
\begin{align*}
\Delta; \Gamma \vdash f : \tau_f \\
\tau_f \equiv \tau_a \to \tau_r
\end{align*}
\]

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

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

▶ E.g., \((\lambda \alpha. \alpha \to \alpha) (\text{int int}) \equiv (\text{int int}) \to (\text{int int})\)

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.

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

▶ Ill-kindred types can’t make well-typed terms get stuck.

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

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:
▶ \( \Delta; \Gamma \vdash e : \tau \quad \tau \equiv \tau' \quad \Delta \vdash \tau' :: \star \) 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”)

Matthew Fluet
CSE-505 2015, Lecture 27
28
Matthew Fluet
CSE-505 2015, Lecture 27
28
Type Checking for System F_ω

Remove non-syntax-directed rules and relations:

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

**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 \( \Delta \vdash \kappa \) and \( \tau \Rightarrow^* \tau' \),
    - then either \( \tau' \) is in (weak-head) normal form (i.e., a type-level “value”) or \( \tau' \Rightarrow \tau'' \).
  - Proofs by Progress and Preservation on kinding and parallel reduction derivations.

---

Matthew Fluet  
CSE-505 2015, Lecture 27  
29  
Matthew Fluet  
CSE-505 2015, Lecture 27  
30

Type Checking for System F_ω

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 \( \Delta \vdash \kappa \) and \( \tau \Rightarrow^* \tau' \),
    - then either \( \tau' \) is in (weak-head) normal form (i.e., a type-level “value”) or \( \tau' \Rightarrow \tau'' \).
  - Proofs 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 \(\Delta, \Gamma, \text{ and } e\), 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 \(\tau \Rightarrow^* \tau'\).
  - 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.”