CSE-505: Programming Languages

Lecture 27 — Higher-Order Polymorphism

Matthew Fluet
2016

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.

Matthew Fluet CSE-505 2016, Lecture 27 5

(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).
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.

(Matplotlib? List Library with ∀/∃

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 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.

Matthew Fluet CSE-505 2016, Lecture 27 6

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.

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

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

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

\[ \ldots \]

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

\[ \tau \equiv \tau' \]

\[ \ldots \]

\[ (\lambda \alpha. \tau_b) \tau_a \equiv \tau_b[\alpha/\tau_a] \]

\[ \ldots \]

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

\[ \tau \equiv \tau' \]

\[ \ldots \]

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

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

\[ \ldots \]
Type-level Expressions

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

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

int → bool  int → Id bool  Id int → bool  Id int → Id bool

Id (int → bool)  Id (Id (int → bool)) ...

Admits “wrong/bad/meaningless” types:

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

Require a “type system” for types:

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

\[ \Delta \vdash \tau f :: \kappa \alpha \Rightarrow \kappa \rho \qquad \Delta \vdash \tau a :: \kappa \alpha \]

\[ \Delta \vdash \tau f \tau a :: \kappa \rho \]

Terms, Types, and Kinds, Oh My

Terms:

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

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

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

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., \( c \)) classify the terms that evaluate to atomic 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)

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., \( \tau \rightarrow \tau \)) 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
- classified by kinds (but not all types have a kind)

Kinds: \( \kappa ::= \ast \mid \kappa \Rightarrow \kappa \)

- kind of proper types \( \ast \) 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

- ⋆
  - the kind of proper types
    - Bool, Bool → Bool, ...
  - ⋆ ⇒ ⋆
    - the kind of (unary) type operators
      - List, Maybe, ...
    - the kind of (binary) type operators
      - Either, Map, ...
  - ((⋆ ⇒ ⋆) ⇒ ⋆) ⇒ ⋆
    - the kind of higher-order type operators
      - taking unary type operators to unary type operators
      - MaybeT, ListT, ...

Matthew Fluet
CSE-505 2016, Lecture 27
12
Matthew Fluet
CSE-505 2016, Lecture 27
12
Kind Examples

- $\star$
  - the kind of proper types
  - $\text{Bool, Bool} \rightarrow \text{Bool, Maybe Bool, Maybe Bool} \rightarrow \text{Maybe Bool, \ldots}$
- $\star \Rightarrow \star$
  - the kind of (unary) type operators
  - $\text{List, Maybe, Map Int, Either (List Bool), \ldots}$
- $\star \Rightarrow \star \Rightarrow \star$
  - the kind of (binary) type operators
  - $\text{Either, Map, \ldots}$
- $(\star \Rightarrow \star) \Rightarrow \star$
  - the kind of (higher-order) type operators
  - taking unary type operators to proper types
  - $\text{???, \ldots}$
- $(\star \Rightarrow \star) \Rightarrow \star \Rightarrow \star$
  - the kind of (higher-order) type operators
  - taking unary type operators to unary type operators
  - $\text{MaybeT, ListT, \ldots}$
System $F_\omega$: Syntax

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

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

Should look familiar:

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

System $F_\omega$: Operational Semantics

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

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

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

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

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

\[
\frac{e a \rightarrow_{\text{cbv}} e'_a}{v f e_a \rightarrow_{\text{cbv}} v f e'_a}
\]

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

- Unchanged! All of the new action is at the type-level.
\[
\begin{align*}
\tau &\equiv \tau' \\
\tau_1 &\equiv \tau_2 \\
\tau_2 &\equiv \tau_1 \\
\tau &\equiv \tau \\
\tau_1 &\equiv \tau_2 \\
\tau_2 &\equiv \tau_3 \\
\tau_1 &\equiv \tau_3 \\
\tau_{a_1} &\equiv \tau_{a_2} \\
\tau_{r_1} &\equiv \tau_{r_2} \\
\tau_{a_1} &\rightarrow \tau_{r_1} \equiv \tau_{a_2} \rightarrow \tau_{r_2} \\
\forall \alpha::\kappa_a. \tau_{r_1} &\equiv \forall \alpha::\kappa_a. \tau_{r_2} \\
\lambda \alpha::\kappa_a. \tau_{b_1} &\equiv \lambda \alpha::\kappa_a. \tau_{b_2} \\
\tau_{f_1} &\equiv \tau_{f_2} \\
\tau_{a_1} &\equiv \tau_{a_2} \\
\tau_{f_1} \tau_{a_1} &\equiv \tau_{f_2} \tau_{a_2} \\
\forall \alpha::\kappa_a. \tau_{b_1} &\equiv \forall \alpha::\kappa_a. \tau_{b_2} \\
\lambda \alpha::\kappa_a. \tau_{f_1} &\equiv \lambda \alpha::\kappa_a. \tau_{f_2} \\
\tau_{f_1} \tau_{a_1} &\equiv \tau_{f_2} \tau_{a_2} \\
\end{align*}
\]

System F\(\omega\): Type System, part 3

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

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

\[
\Delta; \Gamma \vdash c : \text{int} \\
\Delta; \Gamma \vdash x : \tau \\
\Delta \vdash \tau_a :: * \\
\Delta; \Gamma \vdash \lambda x::\kappa_a. e_b : \tau_a \rightarrow \tau_r \\
\Delta; \Gamma \vdash e_f : \tau_a \rightarrow \tau_r \\
\Delta; \Gamma \vdash e_a : \tau_a \\
\Delta; \Gamma \vdash \lambda \alpha::\kappa_a. e_b : \forall \alpha::\kappa_a. \tau_r \\
\Delta; \Gamma \vdash e_f [\tau_a] : \forall \alpha::\kappa_a. \tau_r \\
\Delta; \Gamma \vdash e_f [\tau_a/\alpha] : \forall \alpha::\kappa_a. \tau_r \\
\Delta; \Gamma \vdash e : \tau \quad \tau \equiv \tau' \\
\Delta; \Gamma \vdash e : \tau' \\
\Delta; \Gamma \vdash e : \tau' :: *
\]

Should look familiar:
the full reduction rules of the Lambda Calculus “one level up”
Polymorphic List Library with higher-order $\exists$

List library is an existential package:

pack(λα::⋆.μξ::⋆. unit + (α * ξ), list_library)

as $\exists$L::⋆ ⇒ ⋆. {empty : ∀α::⋆. L α; cons : ∀α::⋆. α → L α → L α; unlist : ∀α::⋆. L α → unit + (α * L α); map : ∀α::⋆. ∀β::⋆. (α → β) → L α → L β; ...

The witness type operator is poly.lists: λα::⋆.μξ::⋆. unit + (α * ξ).

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 Fω 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.
**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 \( \vdash v : \forall \alpha :: \kappa_a. \ \tau_r \), then \( v = \Lambda \alpha :: \kappa_a. \ e_b \).

Complicated by typing derivations that end with:

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

(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:
  - $\tau \iff \tau'$ 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 $\tau \equiv \tau'$ and $\tau \Rightarrow^* \tau'$, 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 $v$:
Canonical Forms

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

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

1. \( v = \lambda x : \tau_a \cdot e_b \).
   We have that \( v = \lambda x : \tau_a \cdot e_b \).

Canonical Forms

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

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

1. \( v = c \).
   Derivation of \( \vdash \alpha \vdash v : \tau_a \rightarrow \tau_r \) must be of the form:

   \[
   \vdash \alpha \vdash c : \tau_a \rightarrow \tau_r \]

   \[
   \vdash \alpha \vdash c : \tau_a \rightarrow \tau_r \]

   \[
   \vdash \alpha \vdash c : \tau_a \rightarrow \tau_r \]

   Therefore, we can construct the derivation \( \vdash \alpha \vdash c : \tau_a \rightarrow \tau_r \).

   We can find a common reduct: \( \vdash \alpha \vdash c : \tau_a \rightarrow \tau_r \) implies \( \vdash \alpha \vdash c : \tau_a \rightarrow \tau_r \).

   Reduction preserves shape: \( \vdash \alpha \vdash c : \tau_a \rightarrow \tau_r \) implies \( \vdash \alpha \vdash c : \tau_a \rightarrow \tau_r \).

   But, \( \vdash \alpha \vdash c : \tau_a \rightarrow \tau_r \) 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.

After weeks of thinking about type systems, kinding seems natural;
but kinding is not required for type safety!

Matthew Fluet CSE-505 2016, Lecture 27

---

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

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

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

$$
\begin{align*}
  e &\rightarrow_{cbv} e' \\
  (\lambda x: \tau. e_b) v_a &\rightarrow_{cbv} e_b[v_a/x] \\
  e_f &\rightarrow_{cbv} e'_f \\
  e_f e_a &\rightarrow_{cbv} e'_f e_a \\
  e_a &\rightarrow_{cbv} e'_a \\
  v_f e_a &\rightarrow_{cbv} v'_f e_a \\
  (\Lambda \alpha. e_b) [\tau_a] &\rightarrow_{cbv} e_b[\tau_a/\alpha] \\
  e_f [\tau_a] &\rightarrow_{cbv} e'_f [\tau_a]
\end{align*}
$$

---

Matthew Fluet CSE-505 2016, Lecture 27

---

Matthew Fluet CSE-505 2016, Lecture 27
This language is type safe.

Check that free type variables of $\tau$ are in $\Delta$, but nothing else.
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 $\Delta_1, \alpha, \Delta_2 \vdash \tau_1 :: \check{\cdot}$ and $\Delta_1 \vdash \tau_2 :: \check{\cdot}$, then $\Delta_1, \Delta_2 \vdash \tau_1[\tau_2/\alpha] :: \check{\cdot}$.
  - 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 $\Delta_1 \vdash \tau_2 :: \check{\cdot}$, 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 $\cdot; \cdot \vdash v : \text{int}$, 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?

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

The typing derivation $\cdot \vdash e : \tau$
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:

\[
\frac{
\Delta \vdash e_f : \tau_f \\
\tau_f \equiv \tau_a \to \tau_r \\
\Delta ; \Gamma \vdash \tau_r
}
\]

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

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?

Why aren’t kinds required for type safety?

Recall statement of type safety:

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

The typing derivation $\cdot \vdash e : \tau$
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., $(\lambda \alpha. \alpha \to \alpha) (\text{int \ int}) \equiv (\text{int \ int}) \to (\text{int \ int})$

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.

Recall the statement of type checking:

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

Two issues:

1. $\Delta; \Gamma \vdash e : \tau \equiv \tau' \quad \Delta \vdash \tau' :: \star$ is a non-syntax-directed rule
2. $\tau \equiv \tau'$ 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\textsubscript{ω}

Remove non-syntax-directed rules and relations:

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

\[\begin{array}{c}
\Delta \vdash x : \tau_a \\
\Delta; \Gamma \vdash e_b : \tau_r \\
\Delta; \Gamma \vdash \lambda x : \tau_a. e_b : \tau_a \rightarrow \tau_r
\end{array}\]

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 \(\tau' \Rightarrow \tau''\).
- Proofs by Progress and Preservation on kinding and parallel reduction derivations.

Kinds are for type checking.

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

Type Checking for System F\textsubscript{ω}

\[\frac{\Delta; \Gamma \vdash e : \tau_f}{\Delta; \Gamma \vdash e [\tau_a] : \tau_f[\tau_a/\alpha]}\]

- If \(\Delta \vdash \tau_a :: \kappa_a, \tau_f \Rightarrow^* \tau_f'\),
  then \(\tau_f' = \forall \alpha :: \kappa_a \cdot \tau_f\).
- \(\Delta; \Gamma \vdash \lambda \alpha. e_b : \forall \alpha :: \kappa_a \cdot \tau_f\)

-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 $\Delta$, $\Gamma$, 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 $\tau \Rightarrow^\downarrow \tau'$.
  - 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.”