CSC2125H
Types and Programming Languages
Polymorphism all the way up!

Ningning Xie
Assistant Professor
Department of Computer Science
University of Toronto

Recall: System F

- From Lecture 4:

$$
\begin{array}{llll}
\text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha.\, \tau \\
\text{Expressions} & e & ::= & x \mid \lambda x{:}\tau.\, e \mid e_1\, e_2 \mid \Lambda \alpha.\, e \mid e\,[\tau] \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau \mid \Gamma, \alpha\ \textit{type}
\end{array}
$$

Higher-order types

- Many programming languages will include ways to define data constructors, e.g.

List[Int]

Higher-order types

- Many programming languages will include ways to define data constructors, e.g.

List[Int]

- What is a List?

Higher-order types

- Many programming languages will include ways to define data constructors, e.g.

List[Int]

- What is a List?

- It is a *type constructor* that takes a type and returns a type!

System Fω

- System Fω is an extension of System F where
  - type expressions include functions from types to types, and
  - terms can abstract (using Λ) over all kinds of types

$$
\begin{array}{llll}
\text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha : \kappa.\tau \mid \lambda\alpha : \kappa.\tau \mid \tau_1\ \tau_2 \\
\text{Expressions} & e & ::= & x \mid \lambda x{:}\tau.\, e \mid e_1\, e_2 \mid \Lambda\alpha : \kappa.\ e \mid e\,[\tau] \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau \mid \Gamma,\ \alpha : \kappa \\
\text{Kinds} & \kappa & ::= & \star \mid \kappa_1 \to \kappa_2
\end{array}
$$

System Fω

- System Fω is an extension of System F where
  - type expressions include functions from types to types, and
  - terms can abstract (using Λ) over all kinds of types

$$
\begin{array}{llll}
\text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha : \kappa.\tau \mid \lambda\alpha : \kappa.\tau \mid \tau_1\ \tau_2 \\
\text{Expressions} & e & ::= & x \mid \lambda x{:}\tau.\, e \mid e_1\, e_2 \mid \Lambda\alpha : \kappa.\ e \mid e\,[\tau] \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau \mid \Gamma,\ \alpha : \kappa \\
\text{Kinds} & \kappa & ::= & \star \mid \kappa_1 \to \kappa_2
\end{array}
$$

- For example, the list type constructor has kind

System Fω

- System Fω is an extension of System F where
  - type expressions include functions from types to types, and
  - terms can abstract (using Λ) over all kinds of types

$$
\begin{array}{llll}
\text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha : \kappa . \tau \mid \lambda \alpha : \kappa . \tau \mid \tau_1 \; \tau_2 \\
\text{Expressions} & e & ::= & x \mid \lambda x{:}\tau .\, e \mid e_1 \, e_2 \mid \Lambda \alpha : \kappa .\; e \mid e\,[\tau] \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau \mid \Gamma,\; \alpha : \kappa \\
\text{Kinds} & \kappa & ::= & \star \mid \kappa_1 \to \kappa_2
\end{array}
$$

- For example, the list type constructor has kind $\star \longrightarrow \star$

Kinding

$$\frac{}{\Gamma_1, \alpha : \kappa, \Gamma_2 \vdash \alpha : \kappa}$$

$$\frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \lambda\alpha : \kappa.\ \tau : \kappa_1 \to \kappa_2}$$

$$\frac{\Gamma \vdash \tau_1 : \kappa_1 \to \kappa_2 \quad \Gamma \vdash \tau_2 : \kappa_1}{\Gamma \vdash \tau_1\ \tau_2 : \kappa_2}$$

$$\frac{\Gamma \vdash \tau_1 : \star \quad \Gamma \vdash \tau_2 : \star}{\Gamma \vdash \tau_1 \to \tau_2 : \star}$$

$$\frac{\Gamma, \alpha : \kappa \vdash \tau : \star}{\Gamma \vdash \forall\alpha : \kappa.\tau}$$

Typing

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash \tau_1 : \star \quad \Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma, \alpha : \kappa \vdash e : \tau}{\Gamma \vdash \Lambda\alpha : \kappa.\ e : \forall\alpha : \kappa.\ \tau}$$

$$\frac{\Gamma \vdash e : \forall\alpha : \kappa.\ \tau \quad \Gamma \vdash \tau' : \kappa}{\Gamma \vdash e[\tau'] : [\tau'/\alpha]\tau}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 =_\beta \tau_2}{\Gamma \vdash e : \tau_2}$$

Forms of parameterization

So far we have seen three parameterization mechanisms:

1.

)

What is a fourth form?

Forms of parameterization

So far we have seen three parameterization mechanisms:

1. a term parameterized by a term: $\lambda x.\ e$
   (= functions from terms to terms)

1.

)

What is a fourth form?

Forms of parameterization

So far we have seen three parameterization mechanisms:

1. a term parameterized by a term: $\lambda x.\ e$
   (= functions from terms to terms)
2. A term parameterized by a type: $\Lambda\alpha.\ e$
   (= polymorphism, System F)
3.

)

What is a fourth form?

Forms of parameterization

So far we have seen three parameterization mechanisms:

1. a term parameterized by a term: $\lambda x.\ e$
   (= functions from terms to terms)
2. A term parameterized by a type: $\Lambda\alpha.\ e$
   (= polymorphism, System F)
3. a type parameterized by a type:
   (= type constructor, System Fω)

What is a fourth form?

Forms of parameterization

So far we have seen three parameterization mechanisms:

1. a term parameterized by a term: $\lambda x.\ e$
   (= functions from terms to terms)
2. A term parameterized by a type: $\Lambda \alpha.\ e$
   (= polymorphism, System F)
3. a type parameterized by a type: $\Lambda \alpha.\ \tau$
   (= type constructor, System Fω)

Forms of parameterization

So far we have seen three parameterization mechanisms:

1. a term parameterized by a term: $\lambda x.\ e$
   (= functions from terms to terms)
2. A term parameterized by a type: $\Lambda\alpha.\ e$
   (= polymorphism, System F)
3. a type parameterized by a type: $\Lambda\alpha.\ \tau$
   (= type constructor, System Fω)

What is a fourth form?

Forms of parameterization

So far we have seen three parameterization mechanisms:

1. a term parameterized by a term: $\lambda x.\ e$
   (= functions from terms to terms)
2. A term parameterized by a type: $\Lambda \alpha.\ e$
   (= polymorphism, System F)
3. a type parameterized by a type: $\Lambda \alpha.\ \tau$
   (= type constructor, System Fω)

4. a type parameterized by a term (= dependent types)

Dependent types in logic

$$\texttt{even}(n) \stackrel{def}{=} n \bmod 2 = 0 \qquad \texttt{odd}(n) \stackrel{def}{=} n \bmod 2 = 1$$

The theorem: if n is even, then (n+1) is odd.

Corresponds to the type: $\forall n : \text{nat}.\ \text{even}(n) \longrightarrow \text{odd}\ (n+1)$

Dependent types for programs

- In Fortran, C or C++, the type of an array t[e] contains
  - a type t: the type of the array elements
  - a ”term” (constant expression) N : the size of the array

  Let’s write array(t, N)

Dependent types for programs

- In Fortran, C or C++, the type of an array t[e] contains
  - a type t: the type of the array elements
  - a ”term” (constant expression) N : the size of the array

  Let’s write array(t, N)

- Lifting the restriction that N is a constant expression, and allowing ourselves to quantify over this N, we can give very precise dependent types to array operations:

Dependent types for programs

- In Fortran, C or C++, the type of an array t[e] contains
  - a type t: the type of the array elements
  - a ”term” (constant expression) N : the size of the array

  Let’s write array(t, N)

- Lifting the restriction that N is a constant expression, and allowing ourselves to quantify over this N, we can give very precise dependent types to array operations:

$$\mathtt{concat} : \forall t. \forall N_1. \forall N_2.\ \mathtt{array}(t, N_1) \to \mathtt{array}(t, N_2) \to \mathtt{array}(t, N_1 + N_2)$$

Dependent types for programs

- Now: are the following array types compatible?

$$\begin{aligned}
\mathtt{array}(t, 6) &\quad \text{and} \quad \mathtt{array}(t, 5+1) \\
\mathtt{array}(t, N+M) &\quad \text{and} \quad \mathtt{array}(t, M+N) \\
\mathtt{array}(t, \mathtt{fact}(N)/\mathtt{fact}(N-1)) &\quad \text{and} \quad \mathtt{array}(t, N) \\
\mathtt{array}(\mathtt{random}(10)) &\quad \text{and} \quad \mathtt{array}(6)
\end{aligned}$$

Dependent types for programs

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \approx \tau_2}{\Gamma \vdash e : \tau_2}$$

- The notion of convertibility is often not just beta-conversion (as in Fω) but includes other equivalences to deal with terms that can occur in the types

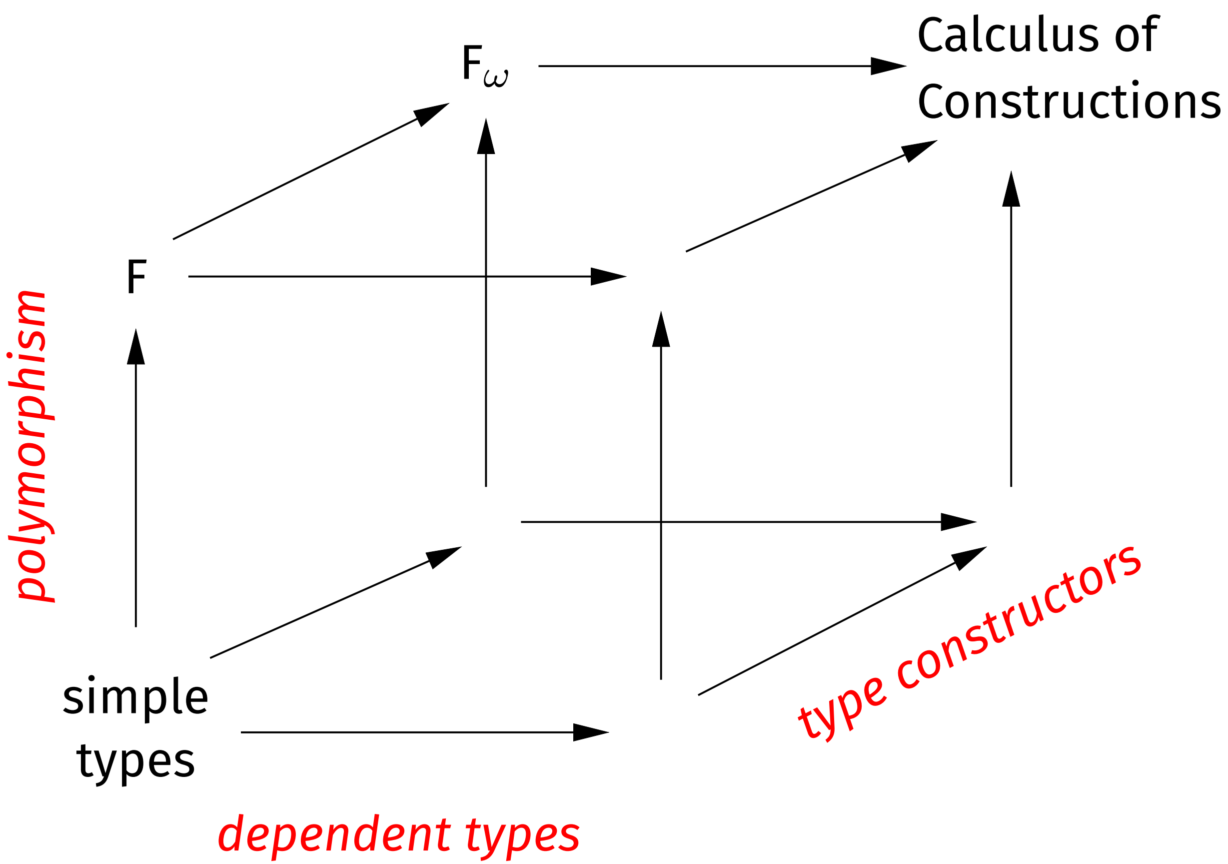
Dependent types for programs

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \approx \tau_2}{\Gamma \vdash e : \tau_2}$$

- The notion of convertibility is often not just beta-conversion (as in Fω) but includes other equivalences to deal with terms that can occur in the types

- Undecidable if the type language is rich enough.

Lambda cube



Barendregt, Henk (1991). "Introduction to generalized type systems". Journal of Functional Programming. 1 (2): 125–154. doi:10.1017/s0956796800020025. hdl:2066/17240. ISSN 0956-7968. S2CID 44757552.

Calculus of construction

- Collapse terms, types, and kinds

Expressions

$$e, \tau, \kappa ::= x \mid \star \mid \Box \mid \lambda x : \tau.\ e \mid e_1\ e_2 \mid \forall x : \tau_1 . \tau_2$$

Contexts $\quad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : \tau$

Typing

- We write s for either $\star \mid \Box$

$$\frac{}{\Gamma \vdash \star : \Box} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash \tau_1 : s_1 \quad \Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 .\ e) : (\forall x : \tau_1 .\ \tau_2)}$$

$$\frac{\Gamma \vdash e_1 : \forall x : \tau_1 .\ \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : [e_2/x]\tau_2}$$

$$\frac{\Gamma \vdash \tau_1 : s_1 \quad \Gamma, x : \tau_1 \vdash \tau_2 : s_2}{\Gamma \vdash (\forall x : \tau_1 .\ \tau_2) : s_2} \qquad \frac{\Gamma \vdash e : \tau_1 \quad \tau_1 =_\beta \tau_2}{\Gamma \vdash e : \tau_2}$$

Encode System Fω

- Notation: $\tau_1 \to \tau_2 \triangleq \forall x : \tau_1.\ \tau_2 \quad \text{if } x \notin \mathsf{fv}(\tau_2)$
- Terms
- Types
- Kinds

Encode System Fω

- Notation: $\tau_1 \to \tau_2 \triangleq \forall x : \tau_1.\ \tau_2 \quad \text{if } x \notin \mathsf{fv}(\tau_2)$
- Terms

$$\lambda x : \tau.\ e \equiv \lambda x : \tau.\ e \qquad\qquad \text{with } \ \tau : \star$$

- Types
- Kinds

Encode System Fω

- Notation: $\tau_1 \to \tau_2 \triangleq \forall x : \tau_1.\ \tau_2 \quad \text{if } x \notin \mathsf{fv}(\tau_2)$
- Terms

$$\lambda x : \tau.\ e \equiv \lambda x : \tau.\ e \qquad \text{with } \ \tau : \star$$

$$\Lambda \alpha : \kappa.\ e \equiv \lambda \alpha : \kappa.\ e \qquad \text{with } \ \kappa : \Box$$

- Types
- Kinds

Encode System Fω

- Notation: $\tau_1 \to \tau_2 \triangleq \forall x : \tau_1.\ \tau_2 \quad \text{if } x \notin \mathsf{fv}(\tau_2)$
- Terms

$$\lambda x : \tau.\ e \equiv \lambda x : \tau.\ e \qquad \text{with } \ \tau : \star$$

$$\Lambda \alpha : \kappa.\ e \equiv \lambda \alpha : \kappa.\ e \qquad \text{with } \ \kappa : \Box$$

- Types

$$\tau_1 \to \tau_2 \equiv \tau_1 \to \tau_2 \qquad \text{with } \ \tau_1, \tau_2 : \star$$

- Kinds

Encode System Fω

- Notation: $\tau_1 \to \tau_2 \triangleq \forall x : \tau_1.\ \tau_2 \quad \text{if } x \notin \mathsf{fv}(\tau_2)$
- Terms

$$\lambda x : \tau.\ e \equiv \lambda x : \tau.\ e \qquad \text{with } \tau : \star$$

$$\Lambda \alpha : \kappa.\ e \equiv \lambda \alpha : \kappa.\ e \qquad \text{with } \kappa : \Box$$

- Types

$$\tau_1 \to \tau_2 \equiv \tau_1 \to \tau_2 \qquad \text{with } \tau_1, \tau_2 : \star$$

$$\forall X : \kappa.\ \tau \equiv \forall X : \kappa.\ \tau \qquad \text{with } \kappa : \Box, \tau : \star$$

- Kinds

Encode System Fω

- Notation: $\tau_1 \to \tau_2 \triangleq \forall x : \tau_1.\ \tau_2 \quad \text{if } x \notin \mathsf{fv}(\tau_2)$
- Terms

$$\lambda x : \tau.\ e \equiv \lambda x : \tau.\ e \qquad \text{with } \tau : \star$$

$$\Lambda \alpha : \kappa.\ e \equiv \lambda \alpha : \kappa.\ e \qquad \text{with } \kappa : \Box$$

- Types

$$\tau_1 \to \tau_2 \equiv \tau_1 \to \tau_2 \qquad \text{with } \tau_1, \tau_2 : \star$$

$$\forall X : \kappa.\ \tau \equiv \forall X : \kappa.\ \tau \qquad \text{with } \kappa : \Box, \tau : \star$$

- Kinds

$$\star \equiv \star \qquad \text{with } \star : \Box$$

Encode System Fω

- Notation: $\tau_1 \to \tau_2 \triangleq \forall x : \tau_1.\ \tau_2 \quad \text{if } x \notin \mathsf{fv}(\tau_2)$
- Terms

$$\lambda x : \tau.\ e \equiv \lambda x : \tau.\ e \qquad \text{with } \tau : \star$$

$$\Lambda \alpha : \kappa.\ e \equiv \lambda \alpha : \kappa.\ e \qquad \text{with } \kappa : \Box$$

- Types

$$\tau_1 \to \tau_2 \equiv \tau_1 \to \tau_2 \qquad \text{with } \tau_1, \tau_2 : \star$$

$$\forall X : \kappa.\ \tau \equiv \forall X : \kappa.\ \tau \qquad \text{with } \kappa : \Box, \tau : \star$$

- Kinds

$$\star \equiv \star \qquad \text{with } \star : \Box$$

$$\kappa_1 \to \kappa_2 \equiv \kappa_1 \to \kappa_2 \qquad \text{with } \kappa_1, \kappa_2 : \Box$$

Summary

- Different forms of parameterizations
- Characterized by the lambda-cube.
- The top of the lambda-cube is the Calculus of Constructions.
- There are variants of dependent type systems, and there also exist more general frameworks.