Introduction to Type Theory — Herman Geuvers

Lecture 2: Polymorphic Type Theory
OPLSS 2026

1 Why polymorphism?

Simple type theory cannot reuse one definition uniformly at many types: the identities $\lambda x : \alpha.x : \alpha \to \alpha$ and $\lambda x : \beta.x : \beta \to \beta$ are distinct typed terms. Polymorphic type theory adds universal types $\forall \alpha.A$ so a single term can act parametrically over all types, as in

$$\forall \alpha.\alpha \to \alpha \qquad \text{and} \qquad \forall \alpha.\forall \beta.\alpha \to \beta \to \alpha.$$

The key restriction is parametricity: this is not overloading by inspecting whether the chosen type is, say, Nat. In fact, this implies that $\lambda x.x$ is the only term of type $\forall \alpha.\alpha \to \alpha$.

2 Full polymorphism

System F-style $\lambda 2$ extends simple types by universal quantification:

$$Typ ::= TVar \mid Typ \to Typ \mid \forall \alpha.Typ.$$

Terms such as $\lambda x.xx$ which could not be typed in STLC can now be assigned the type $\forall \alpha.\alpha \to \forall \alpha.\alpha$.

In Church style, type abstraction and application appear in terms:

$$\frac{\Gamma \vdash M : A \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash \Lambda \alpha.M : \forall \alpha.A} \qquad \frac{\Gamma \vdash M : \forall \alpha.A}{\Gamma \vdash M\,B : A[\alpha := B]}.$$

In Curry style, these type-level operations are implicit. Erasure removes type annotations and type applications from Church terms, and every Curry derivation has some Church term that erases to it.

This is where the algorithmic story changes. Church-style System F has decidable type checking because the term carries enough type information. Curry-style System F type checking is undecidable, a theorem due to Wells.

3 Weak and let polymorphism

ML-style polymorphism restricts $\forall$ to the outside of simple types; type variables are usually left implicitly universally quantified. With this weaker discipline, principal types and decidable type inference are retained. The abstraction rule is also restricted to simple types.

Let polymorphism regains some flexibility:

$$\frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash \mathit{let}\ x = M\ \mathit{in}\ N : B}$$

where A may be polymorphic but the resulting B is simple. This permits a polymorphic definition to be used more than once at different instances, while avoiding full impredicative inference.

4 Second-order formulas as types

The formulas-as-types correspondence for $\lambda 2$ is *constructive* second-order logic. Second-order quantification over propositions corresponds to $\forall \alpha.A$. Classical principles such as Peirce's law,

$$\forall \alpha. \forall \beta. ((\alpha \to \beta) \to \alpha) \to \alpha,$$

are not derivable constructively.

Other connectives can be encoded:

$$\bot := \forall \alpha.\alpha, \quad A \wedge B := \forall \alpha.(A \to B \to \alpha) \to \alpha, \quad A \vee B := \forall \alpha.(A \to \alpha) \to (B \to \alpha) \to \alpha.$$

For example, the first projection from $A \wedge B$ is witnessed by

$$\lambda z : A \wedge B.\, z\, A\, (\lambda x : A. \lambda y : B. x) : (A \wedge B) \to A.$$

Connectives are specified by their elimination behavior. The Church encoding of $A \wedge B$, for instance, packages a pair by saying how to consume both components to produce an arbitrary result.

5 Data types as encodings

Natural numbers are encoded as Church numerals:

$$Nat := \forall \alpha.\alpha \to (\alpha \to \alpha) \to \alpha.$$

The constructors are

$$0 := \Lambda \alpha. \lambda x : \alpha. \lambda f : \alpha \to \alpha. x, \qquad S := \lambda n : Nat. \Lambda \alpha. \lambda x : \alpha. \lambda f : \alpha \to \alpha. f (n\, \alpha\, x\, f).$$

Iteration is built into the encoding. Given $c : A$ and $g : A \to A$, $It\ c\ g := \lambda n : Nat.n\, A\, c\, g$ satisfies $It\ c\ g\ 0 = c$ and $It\ c\ g\ (Sx) = g(It\ c\ g\ x)$.

There are two common successor definitions for Church numerals: applying f outside, $S\, n = \lambda f x.f(nfx)$, or pushing the extra application inside in equivalent presentations. The important invariant is that the resulting numeral applies f one more time than n.

Addition and multiplication follow by iteration:

$$Plus := \lambda m, n : Nat.\, n\, Nat\, m\, S,$$

$$Mult := \lambda m, n : Nat.\, n\, Nat\, 0\, (\lambda y : Nat.Plus\ m\ y).$$

Predecessor is possible but awkward and takes linear time in the represented number because it must simulate primitive recursion using iteration.

There is no efficient predecessor for Church numerals. Untyped Scott numerals make predecessor immediate: $0 = \lambda x.\lambda y.x$, $S = \lambda n.\lambda x.\lambda y.y\, n$, and $Pred = \lambda n.n\, 0\, (\lambda x.x)$. These clauses are not typable as a closed numeral system in the simply typed lambda-calculus: the successor case would force a recursive type equation for the numeral type.

Lists use the analogous fold encoding:

$$List_A := \forall \alpha.\alpha \to (A \to \alpha \to \alpha) \to \alpha.$$

The iteration principle gives definitions such as length and map:

$$length := \lambda \ell : List_A.\ell\, Nat\, 0\, (\lambda a : A.\lambda n : Nat.S\, n),$$

$$Map := \lambda h : A \to B.\lambda \ell : List_A.\ell\, List_B\, Nil\, (\lambda x : A.\lambda k : List_B.Cons\, (h\, x)\, k).$$

6 Initial algebras and terminal coalgebras

The data-type pattern abstracts to functorial type schemes. For a functorial type scheme $F(\alpha)$, a weakly initial algebra can be defined as

$$\mu F := \forall \alpha.(F(\alpha) \to \alpha) \to \alpha.$$

For $g : F(B) \to B$, iteration is $It\ g := \lambda x : \mu F.x\, B\, g$, and the constructor is built so that the initial-algebra diagram commutes.

Dually, weakly terminal coalgebras can be encoded as

$$\nu F := \exists \alpha.(\alpha \times (\alpha \to F(\alpha))).$$

Streams over A are an example, represented by a hidden state type together with a current state, a head function, and a transition function.

The reference from the audience question is Abadi, Cardelli, and Curien's *Formal Parametric Polymorphism* [2]. Section 3.10 states an "erasure conjecture": if two System F terms have the same type in the same environment and identical untyped erasures, then System R proves them related by the identity relation on that type. In symbols, from $E \vdash_F a : A$, $E \vdash_F b : A$, and $erase(a) = erase(b)$, conjecturally $E \vdash_R a : A \; A^* \; b : A$. This addresses the non-injectivity of erasure: the conjecture says that each erasure fiber inside a fixed typing judgment collapses to a single System R equality class.

7 From predicate calculus to dependent types

The pattern is to introduce and eliminate immediately. (For $\forall$ in predicate calculus, $\lambda P.$)

$$\rightarrow \text{ cut elimination / detour elimination.}$$

Rocq creates the λ-term on slide 7 in Lecture 3:

$$\lambda x{:}A.\, \lambda H_1.\, (R\, x\, x) \qquad H\, x\, x \quad H'\, H'$$

8 The idea of the Π-type

A dependent function type collects functions whose result type may depend on the argument:

$$\Pi x{:}A.\, B \;\cong\; \{\, f \;\mid\; \forall a{:}A.\, (f\, a : B) \,\}$$

$$\Pi x{:}A.\, B \;\cong\; \{\, f \;\mid\; \forall a{:}A.\, (f\, a : B[x := a]) \,\}$$

If x does not occur in B (i.e. $x \notin \mathrm{FV}(B)$), the Π-type degenerates to the ordinary arrow:

$$\Pi x{:}A.\, B \;=\; A \rightarrow B.$$

Example.

$$\text{zeros} : \Pi n.\, \mathbb{N} \rightarrow \mathbb{R}^n \qquad \text{zeros } 5 : (0,0,0,0,0)$$

9 Sorts and the conversion rule

The sorts (with their inhabitants one level up):

$$\mathsf{Type} : \Box \;\rightarrow\; \text{(kinds)} \qquad \mathsf{Prop} : *$$

A context is well formed ("put a variable"):

$$A : * \;\Rightarrow\; A \text{ is a type} \qquad A : \Box \;\Rightarrow\; A \text{ is a kind}$$

Conversion rule. If types are β-equivalent they have the same terms (proofs) in them:

$$\frac{P : Q\,(2+3)}{P : Q\,(5)}$$

The conversion rule allows this because $(2+3)$ and 5 are β-equal.

All the rules are syntax-directed. Type checking is subtle because the conversion rule can be applied at any type.

A relation as a domain.

$$A : *, \quad R : A \to A \to * \qquad \big(A = \text{domain},\ R = \text{relation}\big)$$

Reading the flag-style derivation (with $z : A$ in scope):

$$\cfrac{\cfrac{\cfrac{z : A \qquad h : \Pi x, y{:}A.\, R\,x\,y}{h\,z : \Pi y{:}A.\, R\,z\,y}}{h\,z\,z : R\,z\,z}}{h \cdots h\,z\,z : \cdots \qquad (z : A)}$$

10 Lifting Prop to Type

Extending the context:

$$\frac{A : * \qquad P : A \to * \qquad a : A}{\Gamma, \Sigma}$$

$$\text{Type theory} \vdash P\,a \to * : \Box$$

Prop is the top level. Then lift to $T\,p$ (or Πp) to make it inhabitable:

$$\mathsf{Prop} : *$$

$$A \Rightarrow B : \mathsf{Prop} : * \qquad \rightsquigarrow \qquad \text{lifting } T(A \Rightarrow B) : *$$

$$t : T(A \Rightarrow B) \quad \nearrow \ \text{(proofs)}$$

Implication elimination is just the modus ponens rule.

Worked derivation. With $P = \lambda n{:}\mathbb{N}.\, n \geq 0$:

$$\frac{\forall x.\, x \geq 0}{5 \geq 0} \qquad \rightsquigarrow \qquad \frac{T(\forall P)}{T(5 \geq 0)}$$

Abstraction and elimination can build the system:

$$\frac{\Pi x{:}\mathbb{N}.\, T(P\, x)}{\dfrac{T(P\, 5)}{T(5 \geq 0)}}$$

References

[1] J. B. Wells. *Typability and Type Checking in the Second-Order λ-Calculus Are Equivalent and Undecidable*. LICS, 1994.

[2] M. Abadi, L. Cardelli, and P.-L. Curien. *Formal Parametric Polymorphism*. POPL, 1993; also Theoretical Computer Science 121(1–2):9–58, 1993.