1. Why Polymorphism Matters

Strachey (see further reading) identified two types of polymorphic functions:

1. Ad hoc polymorphic ones, like \( + : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \), and \( + : 2 \times 2 \to 2 \) where \( + \) in addition is 00 and exclusive or in 2, so that the meanings are unrelated, and

2. Parametric polymorphic functions, such as

- head and tail of a list \( l \), for \( l \in \text{List } X \)
- where \( X \) could be any type,
- \( \lambda x . x \) for \( I \) any type (identity)
- \( \lambda f . \lambda x . f f x \) for \( I \) any type (doubling)

To these two kinds of polymorphic functions, one could add two more:
3. **Subtype (= Subsort)** polymorphic functions, such as
\[ + : \mathbb{N}^2 \to \mathbb{N}, \quad + : \mathbb{Z}^2 \to \mathbb{Z}, \quad + : \mathbb{Q}^2 \to \mathbb{Q}, \quad + : \mathbb{C}^2 \to \mathbb{C}, \]
where \( \mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C} \) in the number hierarchy from the natural to the complex number.

4. **Dependent polymorphic functions**, such as an array lookup function, `lookup(A, i)`
for a dependent type
\[ \text{Array}(\mathbb{A}) = \sum_{m \in \mathbb{N}} (\mathbb{A}^m), \]
where if \((A, m) \in \text{Array}(\mathbb{A})\), then we must have \(1 \leq i \leq m\) in order for `lookup(A, i)` to be defined.

Support for some of these kinds of polymorphism (poly = multi) (more or fewer) has been given in various semantic frameworks:

First-Order Frameworks like **OSFgH**, **MSFgH**, and rewriting rules provide support for polymorphic functions in the above sense, 1-4. In particular, **Munde** does so, as follows:
1. Ad-hoc polymorphic functions are supported by
   ad-hoc overloading of syntax.

2. Subtype-polymorphic functions are supported by
   subtypes and, even more powerfully, by membership.

3. Both parametric polymorphism and dependent type
   polymorphism are supported by parameterized modules.

Even just for OSEqtl, he prompted Joseph Goguen

to write his "Higher-Order Functions Considered Unnecessary

for Higher-Order Programs" (see further reading). But

since then, Goguen's argument has become stronger for

the following additional reasons not then available:

a. OSEqtl, MEEqtl, and Powerlogic are reflective

   logics (see papers by Clavel, Meseguer & Palomín, TCS, 2007).

This means that one can define not just functions that

take other functions as arguments, but functions that
take entire modules as arguments very easily.

Within the logic itself. It also means that the

notion of parameterized module becomes open

and extensible, not restricted to some fixed

set of theory operations.
6. Since the most powerful higher-order type theories supporting polymorphism such as \( \lambda 2 \) (the polymorphic \( \lambda \)-calculus) and the calculus of constructions (indeed the entire "\( \lambda \)-cube" more on this later) can be represented with 0-representational distance! in both METagl and in Purence logic (see the paper by Stehr and Meseguer in further readys) all those polymorphic type constructions are available at a first-order level at no additional cost anyway.

Higher-Order Framework

1. Dependent-type polymorphic functions are of course supported by Martin-Lof type theory.

2. Parametric polymorphism can be supported in two ways: (a) in limited but useful implicit parametric polymorphism as supported in the ML language, and (b) in a full-blooded explicit parametric polymorphism as exemplified by the Grand-Reynolds polymorphic \( \lambda \)-calculus \( \lambda 2 \).
3. **Sub-type polymorphism** can also be supported, as illustrated by the work of Cardelli & Wegner (ACM Computing Surveys 1985). Note that the links between first-order sub-type polymorphism and higher-order sub-type polymorphism and the categorical logic semantics of both was systematically studied in Martí-Oliet & Meseguer “Inclusiveness and Subtypes I-III” (see course readings).

2. The Polymorphic $\lambda$-Calculus

The Girard-Reynolds polymorphic $\lambda$-calculus, also called System F, and often denoted $\lambda 2$, has the following syntax for types and for terms:

There is an infinite set of type variables, denoted by capital letters $X, Y, Z$, etc. Type expressions are denoted with Greek letter meta-variables such as $\tau, \alpha, \beta$, etc., and have the following grammar:

$$ X | \tau \rightarrow \alpha | \Pi X. \tau $$

Terms are built from variables $x, y, z$, etc. and denoted with meta-variables M, N, U, V, etc.
and hence the following grammar:

\[ \alpha \mid MN \mid \lambda \alpha^x. M \mid M \tau \mid \Lambda X. M \]

The free type variables of a type expression are computed as follows:

\[ \text{ftv}(\alpha) = \alpha \]
\[ \text{ftv}(\tau \rightarrow \alpha) = \text{ftv}(\tau) \cup \text{ftv}(\alpha) \]
\[ \text{ftv}(\Lambda X. \tau) = \text{ftv}(\tau) - \{X\} \]

The only new term constructors are the last two: (i) application of a term \( M \) to a type expression \( \tau \) (which arises \( M \) in of the form \( \Lambda X. N \)), and (ii) type abstraction of a term such as, for example,

- \( \Lambda X. \lambda \alpha^x. \alpha \) (polymorphic identity function)
- \( \Lambda X. \lambda f^x \cdot \lambda \alpha^x. f(\alpha) \) (polymorphic doubling function)

The rest is like in the typed \( \lambda \)-calculus.

Contexts \( \Gamma \) are defined as usual as sequences, \( \Gamma \), of the form

\[ \alpha_1 : \tau_1, \ldots, \alpha_n : \tau_n \]

with \( \alpha_i \neq \alpha_j \) if \( i \neq j \).
\( \alpha \)-conversion is assumed for both type expressions and type variables so that, say,

\[
\Pi X. x \to x \equiv_{\alpha} \Pi Y. Y \to Y
\]

and

\[
\Lambda X. \lambda x^X. x \equiv_{\alpha} \Lambda Y. \lambda y^Y. x
\]

The typing rules of \( \lambda \) are then as follows:

**Axiom**

\[
\Gamma, x : \tau \vdash x : \tau
\]

**App**

\[
\begin{array}{c}
\Gamma \vdash M : \tau \to \alpha \\
\Gamma \vdash N : \tau
\end{array}
\]

\[
\Gamma \vdash MN : \alpha
\]

**Abstr**

\[
\begin{array}{c}
\Gamma, x : \tau \vdash M : \alpha
\end{array}
\]

\[
\Gamma \vdash \lambda x^\tau. M : \tau \to \alpha
\]

**Type-app**

\[
\begin{array}{c}
\Gamma \vdash M : \Pi X. \tau
\end{array}
\]

\[
\Gamma \vdash M \alpha : [X : \alpha] \tau
\]

**Type-abstr**

\[
\begin{array}{c}
\Gamma \vdash M : \tau \\
X \in ftv(\Gamma)
\end{array}
\]

\[
\Gamma \vdash \Lambda X. M : \Pi X. \tau
\]
So, for example, we have

\[ \Lambda X. \; \lambda x^X. \; x : \Pi X. \; X \rightarrow X \]

and

\[ \Lambda X. \; \lambda f^{X \rightarrow X}. \; \lambda x^X. \; f(f \; x) : \Pi X. \; (X \rightarrow X) \rightarrow (X \rightarrow X) \]

The reduction rules are the standard ones in the typed \( \lambda \)-calculus

\begin{align*}
(B) \quad (\lambda x^\tau. \; M) \; N & \rightarrow [x := N] \; M \\
(\eta) \quad \lambda x^\tau. \; M \; x & \rightarrow M \quad \text{if } x \notin \text{fv}(M)
\end{align*}

together with their \( \Lambda \)-versions:

\begin{align*}
(B_{\Lambda}) \quad (\Lambda X. \; M) \; \tau & \rightarrow [x := \tau] \; M \\
(\eta_{\Lambda}) \quad \Lambda X. \; M \; x & \rightarrow M \quad \text{if } X \notin \text{ftv}(M)
\end{align*}

The two key properties of the typed \( \lambda \)-calculus (and of Martin-Löf type theory), namely the Church-Rosser property and strong normalization (all reductions terminate) holds also in \( \Pi \) (see Proofs and Types in the course’s monograph).
3. The Curry-Howard Isomorphism for $\lambda_2$

Since propositions = types, a type expression like

$\Pi x. x \rightarrow x$

is obviously when viewed as a proposition a universal quantification over all propositions.

Logically, this means that we should seek our desired isomorphism to have its logical counterpart in second-order logic, where ordinary first-order variable, like $x, y, z$, etc. range over the given sort (say Nat) but we also have second-order variables $X, Y, Z$, etc., ranging over subsets of the given sort Nat, that is, over propositions or properties of Nat. The paradigmatic example is the Peano axioms of induction:

$$\forall X. (0 \in X \land (\forall x. (x \in X \Rightarrow sx \in X))) \Rightarrow X = \text{Nat}.$$ 

The isomorphism between second-order logic and $\lambda_2$ can then be summarized as follows, where $\tau, \alpha, \beta$, etc. are metavariables ranging over types/formulas, so that $\lambda_2$ terms become proof terms:
where in the Curry-Howard isomorphism we identify $\Pi X. \tau$ with $\forall X. \tau$ and $\rightarrow$ with $\Rightarrow$, and then $ftv(\Gamma')$ becomes $fv^2(\Gamma')$, the set of free second-order variables appearing in context $\Gamma'$. 
4. Categorical Logic Semantics of λ2

Thus as of course several categorical logic semantics for λ2, including, for example:

1. the one by Seely (see further reading) based on Lawvere hyperdoctrines

2. Chapter 8 of Bart Jacobs book "Categorical Logic and Type Theory," Elsevier, 1999

3. J. Meneguine's POPL '89 paper, "Relotioal Model of Polymorphism,"


I will sketch out the main ideas in (3) because:

(a) by building upon the CCC semantics of Kripke's type theory the semantics of a type \( TT \cdot T \) becomes clarified and "demystified," by reducing it to the notion of a type \( TT \times x \sigma \) in Martin-Löf type theory, where \( U \) is a suitable universe over which the variables \( X \) really range;

(b) no extra categorical semantics notions are needed, so this is relatively easy to explain in one lecture;
(c) Seely's model can be understood in relation to (3) as a weaker notion of model (see § 5.6 of (3)).

(d) Many different notions of model of $\lambda_2$, including extensions with fixpoints, $\mu$ models, typed models, and so on, become unified within the framework of (3), once the notion of $\lambda_2$ model is suitably extended (§ 5.2 of (3)), thus the "relating" in the title of (3).

However, this is meant not in lieu of (1), (3), (4) and reference, there, but rather as a stimulus for further research. For example, one of the new contributions of (4) is to give the most full account to date of Reynolds's view of the rôle played by relations in models of $\lambda_2$ to generalize the rôle of homomorphisms in first-order models (see the required reading by Reynolds, "Types, Abstraction and Parametric Polymorphism").

4.1 The Naive Set-Theoretic Universe Model does not Work

A year after publishing the above-mentioned paper, when he was expected to define models of $\lambda_2$ in standard set theory, Reynolds published his "Polymorphism in Set-Theoretic" at a landmark conference on Semantics of Data Types in the Sophia-Antipolis INRIA Campus near Nice (see ref. [47] in (3)).
The key idea of a set-theoretic universe model of $V_2$

is quite intuitive: we would choose a set $U$

such that big enough so that all closed types $T$ of $V_2$

are interpreted as elements $[T] \in U$ and the $\to$

construction is interpreted so that $[T \to x] = [[T] \to [x]] \in U$.

Of course, a typed $x^T(x)$ with $ftv(T) = \{x\}$ will become

a function $[T] = \lambda x \in U. x^T(x) : U \to \overline{U}$, and,

here is the key point, the semantics of $\Sigma X. x(X)$ will

then be the set $\Pi [T] \in U$, where

the set $T = \sum_{x \in U} x$, whose elements are pairs

$(a, x)$ with $a \in x \in U$ and the projection

function $\pi_2 : T \to U$, we can view such a

universe model as a fibration

\[\begin{array}{c}
x \\
\downarrow T_x \\
T_x \\
\downarrow T \\
\end{array}\]

where the set $T_x$ is bijective with the set $x \in U$. 
The "only" problem, however, in that simple cardinality considerations (Lemma 2 in (3)) make clear that $U$ "blows up", so that it must be bigger than any set of any cardinality and therefore naive set-theoretic models of $\lambda_2$ do not exist.

4.2 Categorical Semantics of $\lambda_2$ as a many logics $\lambda_2 \to \lambda T T$

The paper (3) was written at a time when the general logics paper was in draft form (reference [40] in (3)) but used all the general logic ideas. $\lambda_2$ should be viewed not as a single theory, but as a logic with theories $(E, \Xi)$, where $E$ may define new constants and new type constructors, and $\Xi$ give definitional equations. And the Martin–Löf logic $\lambda T T$ is likewise another logic with its own theories. The key ideas in (3) are:

1. That since Martin–Löf type theory is constructive, universes in Martin–Löf type theory are much smaller than in classical set theory and, being constructive (essentially a term model), need not blow up at all.

2. We can give semantics to a $\lambda_2$ theory by mapping it to a $\lambda T T$ theory defining a suitable constructive universe model.
Such a constructive universe can be easily defined by the following rules (where we may as well add products of type):

\[ U \text{- formation} \]

\[
\begin{align*}
 U \text{ set} & \quad \frac{x : U}{T[x] \text{ set}}
\end{align*}
\]

\[ U \text{- introduction} \]

\[
\begin{align*}
 I : U & \\
 T[I] = 1
\end{align*}
\]

\[
\begin{align*}
 x : U & \quad y : U \\
 \frac{x \rightarrow y : U}{T[x \rightarrow y] = T[x] \rightarrow T[y]}
\end{align*}
\]

\[
\begin{align*}
 x : U & \quad y : U \\
 \frac{x \otimes y : U}{T[x \otimes y] = T[x] \otimes T[y]}
\end{align*}
\]

\[
\begin{align*}
 f : U \rightarrow U & \\
 \frac{\prod f : U}{T[\prod f] = \prod_{x \in U} T[f(x)]}
\end{align*}
\]

Of course, if the signature \( \Sigma \) in \( \Lambda \) 2 has other operations and constants such as, e.g., a type-polymorphic function
like \textit{List}[X] with associated constants, say,
\textit{nil}[X] : \textit{List}[X], and \textit{cons}[X] : X \rightarrow \textit{List}[X] \rightarrow \textit{List}[X]

additional rules should be given for \textit{U}, where, among the
\lambda_2\textit{-equations} \textit{E} really define the standard semantics of \textit{lists}, we would
then give \lambda_\Pi\textit{-rules}:

\[
\frac{x : U}{\text{List}(x) : U}
\]
\[
\frac{x : U}{\text{nil}(x) : T[\text{List}(x)]}
\]
\[
\frac{x : U}{\text{cons}(x) : T[x] \rightarrow [T[\text{List}(x)] \rightarrow T[\text{List}(x)]]}
\]

\[
\frac{x : U}{\text{cons}(x) = \lambda y^{T[x]}, \lambda l^{T[\text{List}(x)]}. \langle\langle y, \text{nil}(l)\rangle, s(\text{nil}(l))\rangle}
\]

where we assume that rules introducing \[\Pi, 0, \text{ and } 1\]
have been given as done by Martin-Löf.

The \textit{semantics} in \lambda_\Pi\ of a \lambda_2\textit{-theory} (\Sigma, \Delta) is then defined by the following map of logics

\lambda_2 \rightarrow \lambda_\Pi
Given a λ2-theory \((\Sigma, \mathcal{E})\), we associate to it a λΠ-theory \((\Sigma^0, \mathcal{E}^0 \cup \mathcal{G}_U)\), where \(\Sigma^0\) contains the signature for the universe \(U\), incorporate all other type constructors and constants in \(\Sigma\), \(\mathcal{G}_U\) are the above-mentioned equations for \(T[x:U]\), and types and terms are translated as follows:

\[
\begin{align*}
1 & \mapsto \bar{1} \\
X & \mapsto \bar{X}^U \\
\tau_1 \times \tau_2 & \mapsto \bar{\tau}_1^\circ \times \bar{\tau}_2^\circ \\
\tau_1 \rightarrow \tau_2 & \mapsto \bar{\tau}_1^\circ \rightarrow \bar{\tau}_2^\circ \\
\Pi X. \tau & \mapsto \bar{\Pi} (\lambda X^U. \tau^\circ) \\
F(\tau_1, \ldots, \tau_n) & \mapsto \bar{F}(\tau_1^\circ, \ldots, \tau_n^\circ)
\end{align*}
\]

for any \(F\) in \(\bar{\Sigma}\) such as \(\text{LinA}[x]\).

\[
\begin{align*}
x\tau^\circ & \mapsto \bar{x} \bar{\tau}^\circ \\
\langle s,t \rangle \in \tau_1 \times \tau_2 & \mapsto \langle \bar{s}^\circ, \bar{t}^\circ \rangle : \bar{T}[\tau_1^\circ] \times \bar{T}[\tau_2^\circ] \\
\lambda x\tau : \tau_1 \rightarrow \tau_2 & \mapsto \lambda x \bar{T}[\tau_0^\circ], \bar{t}^\circ : \bar{T}[\tau_0^\circ] \rightarrow \bar{T}[\tau_2^\circ] \\
\Lambda x. \tau : \Pi X. \tau(X) & \mapsto \bar{\lambda} x \bar{X}^U, \bar{t}^\circ : \bar{\Pi}_Y \bar{U}(\bar{Y}^\circ) \rightarrow \bar{T}[\tau(Y)] \\
t \tau : [X := \bar{\tau}] \lambda x(X) & \mapsto \bar{t}^\circ \tau^\circ : \bar{T}[\bar{y}^\circ := \bar{\tau}^\circ] \bar{\lambda} \bar{y}^\circ (Y)
\end{align*}
\]
and where the equations \( E^0 \) are obtained from the equations \( E \) by the above translation, and the equations \( G_U \) are the above-mentioned equality rules for \( U \).

A key theorem in (3) is then:

**Theorem 3.** The map \( \lambda \Sigma \rightarrow \lambda \Pi : (\Sigma, \varepsilon) \mapsto (\Sigma^0, \varepsilon^0 \cup G_U) \)

is a conservative map of logics.

We then define for \((\Sigma, \varepsilon)\) a \( \lambda^2 \)-theory \( \mathcal{LT} \) category of models as:

\[
\text{Mod}(\Sigma, \varepsilon) = \text{Mod}(\Sigma^0, \varepsilon^0 \cup G_U) = \frac{\mathcal{C}(\Sigma^0, \varepsilon^0 \cup G_U)}{\mathcal{LCCC}}
\]

where \( \mathcal{C}(\Sigma^0, \varepsilon^0 \cup G_U) \) is the free \( \mathcal{LCCC} \) generated by \((\Sigma^0, \varepsilon^0 \cup G_U)\). This shows that \( \mathcal{C}(\Sigma^0, \varepsilon^0 \cup G_U) \) is the initial model of \( \text{Mod}(\Sigma, \varepsilon) \) and gives a categorical logic semantics to \( \lambda^2 \) by reduction to the categorical logic semantics of \( \lambda \Pi \).
5. More General Categorical Models of $\lambda^2$

The main limitation of $\lambda T$ models is that, because of the equality types $\mathbb{I}_A$, the category $B/A$ have finite equalizers. But a category with equalizers and fixpoint combinators is intrinsically degenerate. From a language design point of view this means that we cannot:

1. add $\text{letrec}$ to our $\lambda^2$ language, and
2. still have a semantics of our language in $\lambda T$.

The solution to this dilemma is to generalize $\lambda T$ to a weaker/richer logic $\lambda T^\ast$ essentially by discarding the equality types $\mathbb{I}_A$. This can be done by the notion of a relatively cartesian closed category (Def. 10 in (3)), which is really a join $(C, F)$, with $F \leq B$ a category of "diploy ways" such, when restricted to $F$, we get the "effect" of a CCC. In this way, all the $\lambda^2$ models known as of 1989 can be related as $\lambda T^\ast$-models.
6. Where to Go from Here

This finishes the part of the course in which we have been giving semantics in general logics to declarative languages. There are two obvious further directions from here to complete this part so as to answer the following two questions:

1. What other higher-order type theories are of interest to define declarative functional languages and/or prove properties about programs in such languages and

2. How good are MEFqtl or Reunion logic as logical frameworks for all the higher-order logics we have been considering?

The answer to (1) is Barndregt's \( \lambda \)-cube (see attached page) that extends \( \lambda \)2 and \( \lambda P \), where \( \lambda P \) is the Edinburgh Logical Framework LF closely connected to \( \lambda \mathrm{TT} \), and where \( \lambda P w \) is the calculus of constructions.

The answer to (2) is very short: the entire \( \lambda \)-cube can be mapped with 0 representational distance to both MEFqtl and Reunion logic and affords easy implementations in Nandef (see further ready by Stehr & Mossakoua).
where each edge $\rightarrow$ represents the inclusion relation $\subseteq$. This cube will be referred to as the $\lambda$-cube.

The system $\lambda\to$ is the simply typed lambda calculus, already encountered in section 3.2. The system $\lambda 2$ is the polymorphic or second order typed lambda calculus and is essentially the system $F$ of Girard (1972); the system has been introduced independently in Reynolds (1974). The Curry version of $\lambda 2$ was already introduced in Section 4.1. The system $\lambda \omega$ is essentially the system $F \omega$ of Girard (1972). The system $\lambda \Pi$ reasonably corresponds to one of the systems in the family of AUTOMATH languages, see de Bruijn (1980). (A more precise formulation of several AUTOMATH systems can be given as PTSs, see subsection 5.2.) This system $\lambda \Pi$ appears also under the name $LF$ in Harper et al. (1987). The system $\lambda \Pi 2$ is studied in Longo and Moggi (1988) under the same name. The system $\lambda C = \lambda \Pi \omega$ is one of the versions of the calculus of constructions introduced by Coquand and Huet (1988). The system $\lambda \omega$ is related to a system studied by Renardel de Lavalette (1991). The system $\lambda \Pi \omega$ seems not to have been studied before. (For $\lambda \omega$ and $\lambda \Pi \omega$ read: 'weak $\lambda \omega$' and 'weak $\lambda \Pi \omega$' respectively.)

As we have seen in Section 4, the system $\lambda\to$ and $\lambda 2$ can be given also à la Curry. A Curry version of $\lambda \omega$ appears in Giannini and Ronchi (1988) and something similar can probably be done for $\lambda \omega$. On the other hand, no natural Curry versions of the systems $\lambda \Pi$, $\lambda \Pi 2$, $\lambda \Pi \omega$ and $AC$ seem possible.

Now first the systems $\lambda\to$ and $\lambda 2$ à la Church will be introduced in the usual way. Also $\lambda \omega$ and $\lambda \Pi$ will be defined. Then the $\lambda$-cube will be defined.

Fig. 2. The $\lambda$-cube.