Expressiveness and Properties of Theories
There are several ways to assess the expressive power of impredicative polymorphic lambda calculus ($\lambda^{\forall\forall}$).

1. Numeric Functions and Church Numerals

Questions: How to represent numeric functions in $\lambda^{\forall\forall}$?
Representing natural numbers and the type $\text{nat}$.

1. With constant $\text{zero} : \text{nat}$ and $\text{succ} : \text{nat} \to \text{nat}$
   \[ \pi = \text{succ}(\text{succ} \ldots (\text{succ zero}) \ldots) \text{ with } n \text{ occurrences of } \text{succ} \]

2. Without constants $\text{zero}$ and $\text{succ}$
   \[ \pi = \lambda \text{zero} : \text{nat}. \lambda \text{succ} : \text{nat} \to \text{nat}. \text{succ} (\text{succ} \ldots (\text{succ zero}) \ldots) \]

3. Without the type $\text{nat}$
   \[ \pi = \lambda \text{nat}. \lambda \text{zero} : \text{nat}. \lambda \text{succ} : \text{nat} \to \text{nat}. \text{succ} (\text{succ} \ldots (\text{succ zero}) \ldots) \]

4. Church numeral
   \[ \pi = \lambda t. \lambda f : t \to t. \lambda x : t. f^n x \]

5. $\text{nat} = \forall t. (t \to t) \to t \to t$

Other encodings.

1. $\text{bool} = \forall t : t \to t \to t$

2. $\text{true} = \lambda t. \lambda x : t. \lambda y : x$

3. $\text{false} = \lambda t. \lambda x : t. \lambda y : y$

4. zero-test operation: $\pi \text{bool}(\lambda x : \text{bool}. \text{false}) \text{true}$
   zero-test function: $\text{zero}? = \lambda x : \text{nat}. x \text{bool}(\lambda x : \text{bool}. \text{false}) \text{true}$

5. cartesian product
   \[ \sigma \times \tau = \forall t. (\sigma \to \tau \to t) \to t \]

6. The pairing function $\text{pair} : \forall r. \forall s. r \to s \to (r \times s)$ is
   \[ \text{pair} = \lambda r. \lambda s. \lambda x : r. \lambda y : s. \lambda t. \lambda f : r \to s \to t. f x y \]
2. Parametricity

We show some non-parametric functions and demonstrate that adding a simple non-parametric operation to the impredicative calculus causes normalization to fail.

1. Type equality test
   \( TypeEq : \forall s. \forall t. bool \)
   \( \lambda x : s. \lambda y : t. if \ TypeEq \ s \ t \ then \ x \ else \ y : s \to t \to t \) is typable using the equation \( s = t \).

2. Type-condition operation \( TypeCond \)
   \( \Gamma, \sigma = \tau \triangleright M : \rho \)
   \( \Gamma \triangleright TypeCond \ \sigma, \tau \ M \ N : \rho \)

3. Reduction rules for \( TypeCond \)
   \( TypeCond \ \sigma \ M \ N \to M \)
   \( TypeCond \ \sigma, \tau \ M \ N \to N \) \( \sigma, \tau \) distinct, closed

4. Without the side condition on the second rule, the system would be non-confluent, and even inconsistent. Consider the following term:
   \( (\lambda t. TypeCond t s true false) s \)
   \( (\lambda t. TypeCond t s true false) s \to TypeCond s s true false \to true \)
   \( (\lambda t. TypeCond s t true false) s \to (\lambda t. false) s \to false \)

5. Understanding \( TypeCond \) through examples.
   \( mult_N = \lambda x : nat. \lambda y : nat. x \times y : nat \to nat \to nat \)
   \( mult_{N,N} = \lambda x : nat \times nat. \lambda y : nat \times nat. (\text{Proj}_1 x \times \text{Proj}_1 y, \text{Proj}_2 x \times \text{Proj}_2 y) : nat \times nat \to nat \times nat \times nat \)
   Using \( TypeCond \), we can combine the two functions into a single one.
   \( mult = \lambda s. \lambda x : s. \lambda y : s. \\
   TypeCond s nat (mult_N x y) \\
   TypeCond s (nat \times nat)(mult_{N,N} x y) : \forall t. t \to t \to t \)

6. The typing rule using type equations is a problem.
   \( patch_{\forall t. \sigma} : (\forall t. \sigma) \to \forall t. (\sigma \to \forall t. \sigma) \)
   Intuitively, \( patch_{\forall t. \sigma} x \ y \) is like the polymorphic value \( x : \forall t. \sigma \), except at type \( \tau \), where the value is \( y \). This is definable from \( TypeCond \):
   \( patch_{\forall t. \sigma} = \lambda x : \forall t. \sigma. \lambda r : \forall r. [r/t] \sigma. \lambda s. TypeCond s \ r \ y (x s) \)

7. Two reduction rules for \( patch \): (different from the book)
   \( patch_{\forall t. \sigma} M \ \tau \ N \ \tau \to N \)
   \( patch_{\forall t. \sigma} M \ \tau \ N \ \sigma \to M \) \( \sigma, \tau \) distinct, closed

8. Show that \( patch \) violates strong normalization.
   Let \( D \) be a “self-application” function as follows.
Let $Id : apr28.tex,v1.12005/04/2913 : 30 : 20kf ouryExp$ be the polymorphic identity function as usual.

\[
Id = \lambda t.\lambda x : t.x
\]

Combine these two functions at type $\forall t.t \to t$, we have,

\[
X = \text{patch}_{\forall t.t \to t}Id (\forall t.t \to t) D
\]

At last we apply $X (\forall t.t \to t) X$, which has no normal form.

\[
X (\forall t.t \to t) X \to^* D X \to^* X (\forall t.t \to t) X
\]

### 3. No Set-theoretic Model for $\lambda^{-,\forall}$

**Problem Clarification**

This is a relatively subtle point. In any model of $\lambda^{-,\forall}$, we would like:

- Each type denotes a set.
- Each term denotes an element of the appropriate type-set.

We claim that there is a model of this form for $\lambda^{-,\forall}$. However, we can not have a model for $\lambda^{-,\forall}$ satisfying the following two additional conditions:

- The elements of $\sigma \to \tau$ are functions from $\sigma$ to $\tau$.
- The elements of $\forall t.\sigma$ are functions from the collection of types to elements of types.

A more subtle negative result is that we can not have a set-theoretic model satisfying $\sigma \to \tau$ is the set of all functions from $\sigma$ to $\tau$.

**Outline of the Proof** (a detailed proof is huge, involving lots of definitions and lemmas)

1. Show that certain functors are representable in $\lambda^{-,\forall}$.
2. Show that every representable functor $F$ has a “weakly-initial” $F$-algebra.

#### Initial algebra: if $C$ is a class of $\Sigma$-algebra and $A \in C$, then $A$ is initial for $C$, if for every $B \in C$, there is a unique homomorphism $h : A \to B$.

An initial algebra is “typical” in that we may translate from the initial algebra to all other algebra in the class.

#### $F$-algebra: if $C$ is a category and $F : C \to C$ is a functor, then an $F$-algebra is a pair $(a, f)$ consisting of an object $a$ and a morphism $f : F(a) \to a$ in $C$.

3. A contradiction is raised.

Initial $F$-algebra is a solution to the isomorphism $A \cong F(A)$. A weakly initial $F$-algebra $A$ does not provide an isomorphism between $A$ and $F(A)$, but an isomorphism can be constructed using subsets of the semantic interpretations of the $\lambda^{-,\forall}$-types $A$ and $F(A)$ in a set-theoretic model. This leads us a contradiction, since there can be no solution to $A \cong (A \to \text{bool}) \to \text{bool}$ in any model where $\sigma \to \tau$ is all functions from $\sigma$ to $\tau$. 
Data Abstraction and Existential Types

Without going into pragmatic issues associated with data abstraction and program structure, we investigate a general form of data type declaration that may be incorporated into any language with type variables. It may be added to either *predicative* or *impredicative* languages using the general form that we introduce below.

4. Notation We introduce the syntactic form for existential abstraction as

\[ \text{abstype } t \text{ with } x_1 : \sigma_1, \ldots, x_k : \sigma_k \text{ is } \langle \tau, M_1, \ldots, M_k \rangle \text{ in } N \]

where \( M_1, \ldots, M_k \) are implementations of types \( \sigma_1, \ldots, \sigma_k \) that are bound to operations \( x_1, \ldots, x_k \) in \( N \), and \( \tau \) is a concrete type instantiating the abstract type \( t \) in \( \sigma_1, \ldots, \sigma_k \). Note, however, that \( t \) must not appear in the type of \( N \).

5. Example We consider a module implementing streams with two operations, one that selects the first element from a stream, and another that returns the stream of remaining elements.

\[ \text{abstype } \text{stream with} \]
\[ \text{ } s : \text{stream,} \]
\[ \text{ } \text{first} : \text{stream } \rightarrow \text{nat,} \]
\[ \text{ } \text{rest} : \text{stream } \rightarrow \text{stream} \]
\[ \text{is} \]
\[ \text{ } \langle \tau, M_1, M_2, M_3 \rangle \]
\[ \text{in } N \]

Here we suppose some stream \( s \) is given in \( M_1 \), and some implementations of \( \text{first} \) and \( \text{rest} \) are given in \( M_2 \) and \( M_3 \) respectively. Finally, within the scope of \( N \), the stream \( s \) and the functions \( \text{first} \) and \( \text{rest} \) may be used, but the type of \( N \) may not contain \( \text{stream} \).

6. Definition Suppose we have cartesian products. Without loss of generality, we treat a declaration with more than one operation as syntactic sugar.

\[ \text{abstype } t \text{ with } x_1 : \sigma_1, \ldots, x_n : \sigma_n \text{ is } M \text{ in } N \overset{\text{def}}{=} \text{abstype } t \text{ with } y : \sigma_1 \times \cdots \times \sigma_n \text{ is } M \text{ in } [\text{Proj}_1^n y, \ldots, \text{Proj}_n^n y/x_1, \ldots, x_n]N \]

7. Example The aforementioned stream module can be expressed in cartesian products in the \text{abstype} expression as:

\[ \text{abstype } \text{stream with} \]
\[ \text{ } y : \text{stream } \times (\text{stream } \rightarrow \text{nat}) \times (\text{stream } \rightarrow \text{stream}) \]
\[ \text{is} \]
\[ \text{ } \langle \tau, (M_1, M_2, M_3) \rangle \]
\[ \text{in } N' \]

where \( M_1, M_2, M_3 \) are implementations as before, \( N' \) is obtained from \( N \) by substituting \( \text{Proj}_1^3 y \) for all instances of \( s \), \( \text{Proj}_2^3 y \) for all instances of \( \text{first} \), and \( \text{Proj}_3^3 y \) for all instances of \( \text{rest} \).
8. Syntax  To a language with type variables, we extend its syntax with an additional term

\[ M ::= \cdots | \text{abstype } t \text{ with } x : \sigma \text{ is } M \text{ in } M | \langle t = \tau, M : \sigma \rangle \]

where the implementation \( \langle t = \tau, M : \sigma \rangle \) can be thought of as before, but access to representation type \( \tau \) is restricted (when we introduce the rules below). This formulation allows data type implementations to be passed as function arguments, returned as function values, or manipulated in any other way provided by the language.

We also extend type expressions

\[ \sigma ::= \cdots | \exists t. \sigma \]

to include \textit{existential types} of the form \( \exists t. \sigma \). In a predicative language, \( \exists t_1 \ldots \exists t_k. \tau \) would belong to \( U_2 \), assuming \( \tau : U_1 \).

9. Note  Since \( \exists \) is a binding operator, we have \( \exists t. \sigma = \exists s. [s/t] \sigma \), assuming \( s \) does not already occur free in \( \sigma \).

10. Rules for Existential Types  We extend the rules of a language with type variables with the following:

- \( \Gamma \vdash M : [t/\tau] \sigma \)
- \( \Gamma \vdash \langle t = \tau, M : \sigma \rangle : \exists t. \sigma \)  \hspace{1cm} (\exists \text{ Intro})
- \( \Gamma \vdash M : \exists t. \tau \quad \Gamma, x : \tau \vdash N : \sigma \)
- \( \Gamma \vdash ( \text{abstype } t \text{ with } x : \tau \text{ is } M \text{ in } N ) : \sigma \)  \hspace{1cm} (t \text{ not free in } \Gamma \text{ or } \sigma)  \hspace{1cm} (\exists \text{ Elim})

11. Equational Proof

- \( \Gamma \vdash \langle t = \tau, M : \sigma \rangle = \langle s = \tau, M : [s/t] \sigma \rangle : \exists t. \sigma \)  \hspace{1cm} (s \text{ not free in } \sigma)  \hspace{1cm} (\text{Par})
- \( \Gamma \vdash ( \text{abstype } t \text{ with } x : \sigma \text{ is } M \text{ in } N ) = ( \text{abstype } s \text{ with } y : [s/t] \sigma \text{ is } M \text{ in } [y/x][s/t]N ) : \rho \)  \hspace{1cm} (\alpha_\exists)
- \( \Gamma \vdash ( \text{abstype } t \text{ with } x : \sigma \text{ is } \langle t = \tau, M : \sigma \rangle \text{ in } N ) = [M/x][\tau/t]N : \rho \)  \hspace{1cm} (\beta_\exists)
- \( \Gamma \vdash ( \text{abstype } t \text{ with } x : \sigma \text{ is } y \text{ in } \langle t = t, x : \sigma \rangle ) = y : \exists t. \sigma \)  \hspace{1cm} (\eta_\exists)