Introduction to Dependent Type Theory (2/4)
Matthieu Sozeau, Inria Paris & IRIF, University Paris 7

TYPES 2018
August 8th 2018
Ohrid, Macedonia
Lectures

1. Typed Lambda Calculi
2. The Curry-Howard Correspondence
3. Pure Type Systems
4. Dependent Type Theory
The Curry-Howard Correspondence

1. Type checking and inference
2. STLC with products and sums
3. Propositional Logic
4. Beyond propositional logic: System F
5. System $F^\omega$
Typing: $\Gamma \vdash t : \tau$

- **Variable**
  
  \[
  \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}
  \]

- **Abstraction**
  
  \[
  \frac{\Gamma, x : \tau \vdash t : \rho}{\Gamma \vdash \lambda x : \tau. t : \tau \to \rho}
  \]

- **Application**
  
  \[
  \frac{\Gamma \vdash f : \tau \to \rho \quad \Gamma \vdash u : \tau}{\Gamma \vdash fu : \rho}
  \]

**Lemma (Uniqueness of types)**

If $\Gamma \vdash t : \tau$ and $\Gamma \vdash t : \rho$ then $\tau = \rho$.

Typing is syntax-directed, we can hence read these inference rules as an algorithm:

Take $\Gamma, t$ as input and compute $\tau$ such that $\Gamma \vdash t : \tau$ as output.
Type inference algorithm

Take $\Gamma, t$ as input and compute $\tau$ such that $\Gamma \vdash t : \tau$ as output, if any.

Type inference: $\Gamma \vdash t \uparrow \tau$

Variable

\[
\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x \uparrow \tau}
\]

Abstraction

\[
\frac{\Gamma, x : \tau \vdash t \uparrow \rho}{\Gamma \vdash \lambda x : \tau. t \uparrow \tau \rightarrow \rho}
\]

Application

\[
\frac{\Gamma \vdash f \uparrow \alpha \quad \alpha = \tau \rightarrow \rho \quad \Gamma \vdash u \uparrow \beta \quad \beta = \tau}{\Gamma \vdash fu \uparrow \rho}
\]

Properties:

- **Soundness**: $\Gamma \vdash t \uparrow \tau \Rightarrow \Gamma \vdash t : \tau$
- **Completeness**: $\Gamma \vdash t : \tau \Rightarrow \Gamma \vdash t \uparrow \tau$
Type inference algorithm

\[
\begin{align*}
type \Gamma x &= \Gamma(x) \\
type \Gamma (t_1 t_2) &= \text{let } \tau_1 = type \Gamma t_1 \\
&\quad \tau_2 = type \Gamma t_2 \\
&\quad \text{in case } \tau_1 \text{ of} \\
&\quad \quad \tau \rightarrow \tau' \Rightarrow \text{if } \tau = \tau_2 \text{ then } \tau' \text{ else fail} \\
&\quad \quad \text{fail} \\
type \Gamma (\lambda x : \tau. t) &= \tau \rightarrow type (\Gamma[x : \tau]) t
\end{align*}
\]

- Can be extended to take an \textit{untyped} \(\lambda\)-term as input. Requires unification variables for unknown types and a unification algorithm to resolve unification constraints.
1. Type checking and inference

2. STLC with products and sums

3. Propositional Logic

4. Beyond propositional logic: System F

5. System $F^{\omega}$
In untyped $\lambda$-calculus, we could define encodings of datatypes using $\lambda$-terms. However these encodings are essentially untyped, e.g. iszero could apply to any $\lambda$-term, even if not representing a number.

In typed $\lambda$-calculi, we use types to categorize values, introducing new type constructors and value constructors.
Adding a unit type:

\[ \tau, \rho ::= \top \text{ unit type} \]
\[ \tau \to \rho \text{ function type} \]

Unit

\[ \Gamma \vdash () : \top \]

- **One introduction rule**
- **No elimination rule**
The empty type

\[
\begin{align*}
\tau, \rho & ::= \top & \text{unit type} \\
& \quad \bot & \text{empty type} \\
& \quad \tau \to \rho & \text{function type}
\end{align*}
\]

Empty

\[
\begin{align*}
\Gamma \vdash t : \bot \\
\hline
\Gamma \vdash \operatorname{absurd} t : \tau
\end{align*}
\]

- No introduction rule
- One elimination rule
Cartesian products

Pair
\[ \Gamma \vdash t : \tau \quad \Gamma \vdash u : \rho \]
\[ \Gamma \vdash (t, u) : \tau \times \rho \]

Fst
\[ \Gamma \vdash t : \tau \times \rho \]
\[ \Gamma \vdash \text{fst} \, t : \tau \]

Snd
\[ \Gamma \vdash t : \tau \times \rho \]
\[ \Gamma \vdash \text{snd} \, t : \rho \]

We must also explain how to compute with these values.
We extend reduction with new rules (± congruences):

\[ \text{fst} \,(t, u) \rightarrow^\ast_t t \]
\[ \text{snd} \,(t, u) \rightarrow^\ast_u u \]

This encodes the cartesian product of types faithfully.

- Subject reduction holds in the new system \( \lambda \rightarrow^\times \)
- Confluence and termination are preserved as well
The disjoint sum of types can be defined as:

**Left**
\[
\frac{\Gamma |- t : \tau}{\Gamma |- \text{Left} \ t : \tau + \rho}
\]

**Right**
\[
\frac{\Gamma |- t : \rho}{\Gamma |- \text{Right} \ t : \tau + \rho}
\]

**Case**
\[
\frac{\Gamma |- t : \tau + \rho \quad \Gamma, x : \tau |- \ell : \alpha \quad \Gamma, x : \rho |- r : \alpha}{\Gamma |- \text{match} \ t \ \text{with} \ \text{Left} \ x \Rightarrow \ell \mid \ \text{Right} \ x \Rightarrow r \ \text{end} : \alpha}
\]

\[
\text{match Left} \ t \ \text{with} \ \text{Left} \ x \Rightarrow \ell \mid \ \text{Right} \ x \Rightarrow r \ \text{end} \rightarrow_l \ l[x := t]
\]

\[
\text{match Right} \ t \ \text{with} \ \text{Left} \ x \Rightarrow \ell \mid \ \text{Right} \ x \Rightarrow r \ \text{end} \rightarrow_l \ r[x := t]
\]
Subject reduction holds in the new system $\lambda \rightarrow \times +$ One can check that the reduction rules preserve types.

Confluence and termination are preserved as well. The new reduction rules are orthogonal to the existing $\beta$-reduction rule.

Type uniqueness holds as well and type checking is decidable.
The Curry-Howard Correspondence

1. Type checking and inference
2. STLC with products and sums
3. Propositional Logic
4. Beyond propositional logic: System F
5. System $F^{\omega}$
Let’s take a detour to logic, recall the rules of intuitionistic propositional logic.

Formulas:

\[
A ::= P \mid Q \mid R \mid \ldots \text{ propositional variables}
\]

- \(A \Rightarrow A\) implication
- \(A \land A\) conjunction
- \(A \lor A\) disjunction
- \(\top\) truth
- \(\bot\) falsehood
Natural deduction rules, Gentzen 30’s

Provability: $\Gamma \vdash A$

Assumption

$P \in \Gamma$

$\Gamma \vdash P$

→-Intro

$\Gamma, P \vdash Q$

$\Gamma \vdash P \Rightarrow Q$

→-Elim

$\Gamma \vdash P \Rightarrow Q$

$\Gamma \vdash P$

$\Gamma \vdash Q$
Natural deduction rules, Gentzen 30’s

Provability: $\Gamma \vdash A$

Assumption

$P \in \Gamma$

$\Gamma \vdash P$

$\rightarrow$-Intro

$\Gamma, P \vdash Q$

$\Gamma \vdash P \Rightarrow Q$

$\rightarrow$-Elim

$\Gamma \vdash P \Rightarrow Q$

$\Gamma \vdash P$

$\Gamma \vdash Q$

$\land$-Intro

$\Gamma \vdash P$

$\Gamma \vdash Q$

$\Gamma \vdash P \land Q$

$\land$-ElimL

$\Gamma \vdash P \land Q$

$\Gamma \vdash P$

$\Gamma \vdash Q$

$\land$-ElimR

$\Gamma \vdash P \land Q$

$\Gamma \vdash P$

$\Gamma \vdash Q$
Natural deduction rules, Gentzen 30’s

Provability: $\Gamma \vdash A$

- **Assumption**
  
  \[
  \frac{P \in \Gamma}{\Gamma \vdash P}
  \]

- **→-Intro**
  
  \[
  \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}
  \]

- **→-Elim**
  
  \[
  \frac{\Gamma \vdash P \Rightarrow Q \quad \Gamma \vdash P}{\Gamma \vdash Q}
  \]

- **∧-Intro**
  
  \[
  \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \land Q}
  \]

- **∧-ElimL**
  
  \[
  \frac{\Gamma \vdash P \land Q}{\Gamma \vdash P}
  \]

- **∧-ElimR**
  
  \[
  \frac{\Gamma \vdash P \land Q}{\Gamma \vdash Q}
  \]

- **+-IntroL**
  
  \[
  \frac{\Gamma \vdash P}{\Gamma \vdash P \lor Q}
  \]

- **+-IntroR**
  
  \[
  \frac{\Gamma \vdash Q}{\Gamma \vdash P \lor Q}
  \]

- **+-ElimL**
  
  \[
  \frac{\Gamma \vdash P \lor Q \quad \Gamma, P \vdash R \quad \Gamma, Q \vdash R}{\Gamma \vdash R}
  \]
Intuitionistic logic differs from classical logic:

The BHK interpretation:

- A proof of $P \lor Q$ is either a proof of $P$ or a proof of $Q$.
- A proof of $P \land Q$ is a pair of a proof of $P$ and a proof of $Q$.
- A proof of $P \Rightarrow Q$ is an algorithm transforming a proof of $P$ into a proof of $Q$.
- The is no proof of $\bot$, and there is a single proof of $\top$.

This clearly cannot apply to excluded-middle: $\forall P, P \lor \neg P$ which is part of classical logic.
There is a clear correspondence between proofs of propositional logic and typing derivation of $\lambda \rightarrow \times +$!

<table>
<thead>
<tr>
<th>$\lambda \rightarrow \times +$</th>
<th>Intuitionistic Propositional Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Types</strong></td>
<td><strong>Formulas</strong></td>
</tr>
<tr>
<td>$A \rightarrow B, A \times B, A + B$</td>
<td>$A \Rightarrow B, A \land B, A \lor B$</td>
</tr>
<tr>
<td><strong>Typing:</strong> $\Gamma \vdash t : \tau$</td>
<td><strong>Provability:</strong> $\Gamma \vdash A$</td>
</tr>
<tr>
<td><strong>Type-correct (\lambda)-terms</strong></td>
<td><strong>Proofs</strong></td>
</tr>
<tr>
<td>Example: $x : \alpha \vdash x : \alpha$</td>
<td>$A \vdash A$</td>
</tr>
<tr>
<td>$\vdash \lambda x : \alpha.x : \alpha \rightarrow \alpha$</td>
<td>$\vdash A \Rightarrow A$</td>
</tr>
</tbody>
</table>
The Curry-Howard Correspondence

<table>
<thead>
<tr>
<th>$\lambda$-calculus</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programs</td>
<td>Proofs</td>
</tr>
<tr>
<td>Strong Normalization</td>
<td>Cut Elimination</td>
</tr>
<tr>
<td>Syntax directedness</td>
<td>Subformula property</td>
</tr>
<tr>
<td>Type Inhabitation</td>
<td>Provability</td>
</tr>
</tbody>
</table>

The correspondence allows result to flow between programming languages and logics.

History:

- 1950s: Curry
- 1970: Lambek - Cartesian Closed Categories, a model of $\lambda \rightarrow$
- 1970s: de Bruijn - AUTOMATH
- The Brouwer-Heiting-Kolmogorov interpretation of intuitionistic connectives is formalized by this correspondence.
Subformula property

Definition

A proof is in normal form if the corresponding λ-term is in normal form.

This means that the derivation cannot contain →-Intro/→-Elim combinations, which would correspond to reducible β-redexes in the normal λ-term.

Lemma (Subformula property)

In a proof of $\Gamma \vdash A$ in normal form, only subformulas of $\Gamma$ and $A$ occur.

Proof sketch.

By analysis of the possible normal form derivations, →-Elim’s premise can only come from an assumption $A_1 \rightarrow \ldots A_n \in \Gamma$. This corresponds to the fact that the heads of normal term applications are always variables.
Example: Inhabitation

Theorem (Decidability of IPL)

\[ \Gamma \vdash A \text{ is decidable} \]

Proof sketch.

We can search for a derivation in normal form. By the subformula property, we can do a backtracking proof search trying to prove \( A \) from assumptions in \( \Gamma \) and know we never have to invent a formula not already present in \( \Gamma, A \).

Corollary

*Type inhabitation* \( \exists t, \Gamma \vdash t : \tau \text{ is decidable for } \text{STLC} \text{ (even with procuts and sums)}.*
Terms are not in one-to-one correspondance with formulae:

\[ f : \alpha \rightarrow \alpha, x : \alpha \vdash x : \alpha \]
\[ f : \alpha \rightarrow \alpha, x : \alpha \vdash fx : \alpha \]
\[ f : \alpha \rightarrow \alpha, x : \alpha \vdash f(fx) : \alpha \]
\[ x : \alpha, y : \alpha \vdash x : \alpha \]
\[ x : \alpha, y : \alpha \vdash y : \alpha \]

\[ \Rightarrow \text{terms carry more "intensional" information} \]
The Curry-Howard Correspondence

1. Type checking and inference
2. STLC with products and sums
3. Propositional Logic
4. Beyond propositional logic: System F
5. System $F^\omega$
Quantification on types

System F is an extension of $\lambda \rightarrow$ with type quantification.

Types:

$\tau, \rho ::= \alpha, \beta \in \mathcal{V}$  \hspace{1cm} type variables
$\tau \rightarrow \rho$  \hspace{1cm} function type
$\forall \alpha.\tau$  \hspace{1cm} type quantification

Terms:

$t, u ::= x \in \mathcal{V} | \lambda x : \tau.t | t u$
$\Lambda \alpha.t$  \hspace{1cm} type abstraction
$t \tau$  \hspace{1cm} type application

Type Abstraction

\[
\frac{\Gamma \vdash t : \tau \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash \Lambda \alpha.t : \forall \alpha.\tau}
\]

Type Application

\[
\frac{\Gamma \vdash f : \forall \alpha.\tau}{\Gamma \vdash f \rho : \tau[\alpha := \rho]}
\]

We extend beta-reduction:

\[(\Lambda \alpha.t) \tau \rightarrow_\beta t[\alpha := \tau] \]
As a logic: second order intuitionistic logic with universal quantifier.

As a programming language: polymorphic lambda calculus. At the basis of the ML family of languages.

Type polymorphism allows to express generic programs that work at any type.

Example:

\[ \text{id} ::= (\Lambda \alpha. \lambda x : \alpha.x) : \forall \alpha. \alpha \to \alpha \]

This identity function can be applied to any type.
Impredicative encodings

\[ \text{id} : \forall \alpha.\alpha \rightarrow \alpha \]

It can even be applied at its own type:

\[ \text{id} (\forall \alpha.\alpha \rightarrow \alpha) : (\forall \alpha.\alpha \rightarrow \alpha) \rightarrow (\forall \alpha.\alpha \rightarrow \alpha) \]

This is impredicativity. A quantifier can be instantiated by something larger than the type it appears in itself.

This allows to recover Church encodings of data structures in a typed setting, e.g.:

\[
\begin{align*}
\mathbb{B} &::= \forall \alpha.\alpha \rightarrow \alpha \rightarrow \alpha \\
\text{true} &::= \Lambda \alpha.\lambda x y : \alpha.x \\
\text{false} &::= \Lambda \alpha.\lambda x y : \alpha.y \\
\text{ifte} &::= \forall \alpha.\mathbb{B} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha \\
&::= \Lambda \alpha.\lambda b l r : b \alpha l r
\end{align*}
\]
Encoding products

Cartesian products can be derived using the following definition:

\[ \alpha \times \beta ::= \forall \delta. (\alpha \to \beta \to \delta) \to \delta \]

pair ::= \forall \alpha \beta. \alpha \to \beta \to \alpha \times \beta

::= \Lambda \alpha \beta. \lambda (x : \alpha) (y : \beta). \Lambda \delta. \lambda f. f x y

fst ::= \Lambda \alpha \beta. \lambda p. \alpha (\lambda (x : \alpha) (y : \beta). x)

snd ::= \Lambda \alpha \beta. \lambda p. \beta (\lambda (x : \alpha) (y : \beta). y)

We can calculate:

\[
\begin{align*}
\text{fst} (\text{pair} \ l \ r) & \to_\beta (\lambda p. p \alpha (\lambda x y. x)) ((\lambda x y. \Lambda \delta. \lambda f. f x y) \ l \ r) \\
& \to_\beta (\lambda p. p \alpha (\lambda (x : \alpha) (y : \beta). x)) (\Lambda \delta. \lambda f. f \ l \ r) \\
& \to_\beta (\Lambda \delta. \lambda f. f \ l \ r) \alpha (\lambda (x : \alpha) (y : \beta). x) \\
& \to_\beta (\lambda f. f \ l \ r) (\lambda (x : \alpha) (y : \beta). x) \\
& \to_\beta (\lambda (x : \alpha) (y : \beta). x) \ l \ r) \\
& \to_\beta l
\end{align*}
\]
The empty type

\[
\bot ::= \forall \alpha. \alpha
\]

absurd ::= \forall \alpha. \bot \to \alpha

absurd := \Lambda \alpha. \lambda f. f \alpha

- The empty type again has no introduction rule (like falsehood in logic).

- Thanks to the strong normalization property of System F, one can show consistency (not every type is inhabited):

**Theorem (Consistency)**

*There exists no \( t \) such that \( \vdash t : \bot \)*

SN is proven using an extension of the reducibility candidates/logical relations method using a sophisticated domain of interpretation (Girard’72).
In general, one can construct free inductive structures built from constructors and derive their elimination rules.

Girard and Lafont’s Proofs and Types (available on the web) give a general construction.

Natural numbers, lists, arbitrary branching trees can be derived this way etc...
Existential quantification can be derived from universal quantification:

$$\exists \alpha. \tau ::= \forall \beta. (\forall \alpha. (\tau \to \beta)) \to \beta$$

It validates the rules:

**Exists-Intro**

$$\Gamma \vdash t : \tau \quad \Gamma \vdash u : \rho[\alpha := \tau]$$

$$\Gamma \vdash \langle t, u \rangle : \exists \alpha. \rho$$

**Exists-Elim**

$$\Gamma \vdash t : \exists \alpha. \tau \quad \Gamma, x : \tau[\alpha := \rho] \vdash u : \beta \quad \rho \notin FV(\Gamma, \beta)$$

$$\Gamma \vdash \text{unpack } t \text{ as } \langle \rho, x \rangle \text{ in } u : \beta$$

**Example**: \(\langle \top, ((), \lambda x : \top. x) \rangle : \exists \alpha. (\alpha \times (\alpha \to \alpha))\)

**Exercise**: define the introduction and elimination encodings.
Abstract data types

From a programming viewpoint, existential types give abstract data types. E.g. we can represent an abstract type with an ordering function monoid as follows:

\[
\text{OrderedType} ::= \exists \alpha. \alpha \to \alpha \to \text{comparison}
\]

From this one could derive an implementation of a set structure on this type, generically. Such things are called functors in the literature on modules:

\[
\text{MakeSet} : \text{OrderedType} \to \text{Set}
\]

Actually, one can translate away the ML module system to an extension of System F ($F^\omega$) using existential types (see F-ing modules by Rossberg, Russo and Dreyer).
System F was independently discovered by Reynolds and Girard
Reynolds motivation was \textit{parametricity}
Idea: polymorphic functions are parametric on their type argument and behave uniformly.
System F was independently discovered by Reynolds and Girard

Reynolds motivation was *parametricity*

Idea: polymorphic functions are parametric on their type argument and behave uniformly.

Parametricity can be defined as a translation from types to relations on terms such that:

**Theorem (Parametricity)**

*If* $\Gamma \vdash t : \tau$ *then* $\llbracket \tau \rrbracket t t$

\[
\begin{align*}
\llbracket B \rrbracket_{\Gamma} x y & \leftrightarrow x = y \\
\llbracket \tau \rightarrow \rho \rrbracket_{\Gamma} f g & \leftrightarrow \forall x, y, \llbracket \tau \rrbracket_{\Gamma} x y \rightarrow \llbracket \rho \rrbracket (f x) (g y) \\
\llbracket \forall \alpha.\tau \rrbracket_{\Gamma} f g & \leftrightarrow \forall \alpha R, \llbracket \tau \rrbracket_{\Gamma, \alpha \mapsto R} (f \alpha) (g \alpha)
\end{align*}
\]

Parametricity is again a *logical relation*. 
The parametricity theorem for the polymorphic identity says:

\[
\forall \alpha. \alpha \rightarrow \alpha \] \text{id id} = \forall \tau R (x y : \tau), R x y \rightarrow R (\text{id } \tau x) (\text{id } \tau y)
\]

- Without looking at the implementation of id, we now it must transport related arguments to related results!
- These are the so called “Theorems for free!” of Wadler.
The Curry-Howard Correspondence

1. Type checking and inference
2. STLC with products and sums
3. Propositional Logic
4. Beyond propositional logic: System F
5. System $F^\omega$
System $F^\omega$

System F can be extended to handle type constructors:

**Kinds:**

\[
\kappa, \nu ::= * \quad \text{type}
\]

\[
\kappa \to \nu \quad \text{type constructor}
\]

**Types:**

\[
\tau, \rho ::= \mathcal{V} \mid \tau \to \rho
\]

\[
\forall \alpha : \kappa.\tau \quad \text{type quantification}
\]

\[
\lambda \alpha : \kappa.\tau \quad \text{type level abstraction}
\]

\[
\tau \rho \quad \text{type level application}
\]

**Terms:**

\[
t, u ::= x \in \mathcal{V} \mid \lambda x : \tau.\text{t} \mid \text{t u}
\]

\[
\Lambda \alpha : \kappa.\text{t} \quad \text{type abstraction}
\]

\[
t \tau \quad \text{type application}
\]
System $F^\omega$

**Kinding:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>$(\alpha : \kappa) \in \Gamma$</td>
<td>$\Gamma \vdash_k \alpha : \kappa$</td>
</tr>
<tr>
<td>Abstraction</td>
<td>$\Gamma, \alpha : \kappa \vdash_k \tau : \nu$</td>
<td>$\Gamma \vdash_k \lambda \alpha : \kappa. \tau : \forall \alpha : \kappa. \nu$</td>
</tr>
<tr>
<td>Application</td>
<td>$\Gamma \vdash_k f : \kappa \rightarrow \nu$</td>
<td>$\Gamma \vdash_k \tau : \kappa$</td>
</tr>
<tr>
<td></td>
<td>$\Gamma \vdash_k f , \tau : \nu$</td>
<td></td>
</tr>
<tr>
<td>Typing: annotated type abstraction and application.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Type Abstraction | $\Gamma, \alpha : \kappa \vdash t : \tau$ | $\Gamma \vdash \Lambda \alpha : \kappa. t : \forall \alpha : \kappa. \tau$ |
| Type Application| $\Gamma \vdash f : \forall \alpha : \kappa. \tau$ | $\Gamma \vdash_k \rho : \kappa$ |
|                |                                  | $\Gamma \vdash f \, \rho : \tau[\alpha := \rho]$ |

We extend beta-reduction to types:

$$(\lambda \alpha : \kappa. \tau) \, \nu \rightarrow_\beta \tau[\alpha := \nu]$$
▶ Function space, product and sums can be given types:

\[ \to, \times, + : \ast \to \ast \to \ast \]

▶ Polymorphic datatypes have a first-class status, e.g.:

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>( \ast \to \ast )</td>
</tr>
<tr>
<td>nil</td>
<td>( \forall \alpha : \ast . \text{list} \alpha )</td>
</tr>
<tr>
<td>cons</td>
<td>( \forall \alpha : \ast . \alpha \to \text{list} \alpha \to \text{list} \alpha )</td>
</tr>
<tr>
<td>list_rec</td>
<td>( \forall \alpha \beta : \ast . \beta \to (\alpha \to \text{list} \alpha \to \beta \to \beta) \to (\text{list} \alpha \to \beta) )</td>
</tr>
</tbody>
</table>
As a logic: allows abstraction on predicates, relations, connectives, e.g.:

\[
\text{symmetric} : \quad (\ast \to \ast \to \ast) \to \ast
\]
\[
\text{symmetric} ::= \quad \lambda P : \ast \to \ast \to \ast. \forall \alpha \beta : \ast. P \alpha \beta \to P \beta \alpha
\]
\[
\text{prodsym} : \quad \text{symmetric} (\times)
\]
\[
\text{prodsym} ::= \quad \lambda \alpha \beta : \ast. \lambda p : \alpha \times \beta. (\text{snd} \ p, \text{fst} \ p)
\]

As a programming language: Allows to quantify on type constructors, e.g.:

\[
\text{hasmap} : \quad (\ast \to \ast) \to \ast
\]
\[
::= \quad \lambda P : \ast \to \ast. \forall (a \beta : \ast). (\alpha \to \beta) \to P \alpha \to P \beta
\]
\[
\text{map} : \quad \text{hasmap list}
\]
\[
\equiv_{\beta} \quad \forall (a \beta : \ast). (\alpha \to \beta) \to \text{list} \ \alpha \to \text{list} \ \beta
\]
Metatheoretical Properties:

- Subject reduction and strong normalization hold (proof based on candidates of reducibility, extending the System F proof)
- Consistent as a logic.
- Impredicative: $(\forall \alpha : \ast. \alpha) : \ast$. There exists predicative variants stratifying the kinds.
- Enjoys parametricity as well.
A core calculus for functional programming languages, esp. Haskell (extended with type-equalities).

Abstraction on type constructors is essential to model monads:

\[
\text{isMonad} (m : \alpha \rightarrow \beta) ::= (\forall \alpha. \alpha \rightarrow m \alpha) \times (\forall \alpha \beta. \beta \rightarrow (\alpha \rightarrow m \beta) \rightarrow m \beta)
\]

Type-inference, annotating a term to produce a valid $F^\omega$ term is undecidable. Rank-n polymorphism requires order-n unification, which is undecidable from $n > 1$ (Huet, Levy).

The restricted Hindley-Milner system (rank-1 polymorphism) is at the basis of ML.
Typed lambda calculi and logics are strongly related

$\lambda$-terms can be used to represent proofs in various logics

System F adds polymorphism to simply-typed lambda calculus and allow the Church encodings of usual datastructures to be typed.

System $F^\omega$ permits abstraction on these type constructors.

Parametricity can be used to derive theorems from the types of terms.

Next up:

Pure Type Systems: a uniform framework for dealing with term/type/type constructor quantifications.

Dependent types and the Calculus of Constructions