Inductive types

(§6.4, p76→)
PLC-style encoding of algebraic datatypes

booleans $\forall \alpha (\alpha \rightarrow \alpha \rightarrow \alpha)$
natural numbers $\forall \alpha (\alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha)$

etc
Calculus of Constructions

is the Pure Type System $\lambda C$, where $C = (S_C, A_C, R_C)$ is the PTS specification with

\[ S_C \triangleq \{ \text{Prop}, \text{Set} \} \]
\[ A_C \triangleq \{ (\text{Prop}, \text{Set}) \} \]
\[ R_C \triangleq \{ (\text{Prop}, \text{Prop}, \text{Prop})^1, (\text{Set}, \text{Prop}, \text{Prop})^2, (\text{Prop}, \text{Set}, \text{Set})^3, (\text{Set}, \text{Set}, \text{Set})^4 \} \]

1. $\text{Prop}$ has implications, $\phi \to \psi = \Pi x : \phi (\psi)$ (where $\phi, \psi : \text{Prop}$ and $x \notin \text{fv}(q)$).

2. $\text{Prop}$ has universal quantifications over elements of a type, $\Pi x : A (\phi(x))$ (where $A : \text{Set}$ and $x : A \vdash \phi(x) : \text{Prop}$).
   N.B. $A$ might be $\text{Prop}$ ($\lambda 2 \subseteq \lambda C$).

3. $\text{Set}$ has types of function dependent on proofs of a proposition, $\Pi x : p (A(x))$ (where $p : \text{Prop}$ and $x : p \vdash A(x) : \text{Set}$).

4. $\text{Set}$ has dependent function types, $\Pi x : A (B(x))$ (where $A : \text{Set}$ and $x : A \vdash B(x) : \text{Set}$).
PLC-style encoding of algebraic datatypes in \( \lambda C \):

- Booleans: \( \top p : Prop (p \to p \to p) \)
- Natural numbers: \( \top p : Prop (p \to (p \to p) \to p) \)
- Etc.
PLC-style encoding of algebraic datatypes in $\lambda C$

\[\text{bool} \triangleq \neg \neg p : \text{Prop} (p \rightarrow p \rightarrow p)\]
\[\text{nat} \triangleq \neg \neg p : \text{Prop} (p \rightarrow (p \rightarrow p) \rightarrow p)\]

have $\emptyset \vdash \text{bool} : \text{Prop}$

and $\emptyset \vdash \text{nat} : \text{Prop}$

and $\emptyset \vdash t : \text{bool} \leftrightarrow \text{nat}$ for some $t$

How can we get bool, nat, etc. of type Set?
PLC-style encoding of algebraic datatypes in $\lambda C$

\[ \text{nat} \triangleq \top p : \text{Prop} ( p \rightarrow (p \rightarrow p) \rightarrow p ) \]

\[ \top x : \text{Set} ( x \rightarrow (x \rightarrow x) \rightarrow x ) \]

is not typeable in $\lambda C$

(needs a sort $s$ with $\text{Set} : s$)

How can we get $\text{bool, nat, etc}$ of type $\text{Set}$?
The Pure Type System $\lambda U$

is given by the PTS specification $U = (S_U, A_U, R_U)$, where:

$S_U \triangleq \{ \text{Prop, Set, Type} \}$

$A_U \triangleq \{ (\text{Prop, Set}), (\text{Set, Type}) \}$

$R_U \triangleq \{ (\text{Prop, Prop, Prop}), (\text{Set, Prop, Prop}), (\text{Type, Prop, Prop}),$

$\quad (\text{Set, Set, Set}), (\text{Type, Set, Set}) \}$
The Pure Type System $\lambda U$

is given by the PTS specification $U = (S_U, A_U, R_U)$, where:

$S_U \triangleq \{ \text{Prop, Set, Type} \}$

$A_U \triangleq \{ (\text{Prop, Set}), (\text{Set, Type}) \}$

$R_U \triangleq \{ (\text{Prop, Prop, Prop}), (\text{Set, Prop, Prop}), (\text{Type, Prop, Prop}),$

$(\text{Set, Set, Set}), (\text{Type, Set, Set}) \}$

**Theorem** (Girard). $\lambda U$ is logically inconsistent: every legal proposition $\Gamma \vdash P : \text{Prop}$ has a proof $\Gamma \vdash M : P$. (In particular, there is a proof of falsity $\bot \triangleq \Pi p : \text{Prop} (p)$.)
Inductive types (informally)

An inductive type is specified by giving

- *constructor functions* that allow us to inductively generate data values of that type
  (Some restrictions on how the inductive type appears in the domain type of constructors is needed to ensure termination of reduction and logical consistency.)

- *eliminators* for constructing functions on the data

- *computation rules* that explain how to simplify an eliminator applied to constructors.
Extending $\lambda C$ with an inductive type of natural numbers

Pseudo-terms

\[ t ::= \cdots \mid \text{Nat} \mid \text{zero} \mid \text{succ} \mid \text{elimNat}(x.t) t t \]
Extending $\lambda C$ with an inductive type of natural numbers

Pseudo-terms

\[ t ::= \cdots | Nat \mid zero \mid succ \mid \text{elimNat}(x.t) \ t \ t \]

Typing rules

- formation: $\Diamond \vdash Nat : Set$
- introduction: $\Diamond \vdash zero : Nat \quad \Diamond \vdash succ : Nat \to Nat$
- elimination:

\[
\frac{
\Gamma, x : Nat \vdash A(x) : s \quad \Gamma \vdash M : A(\text{zero}) \\
\Gamma \vdash F : \Pi x : Nat (A(x) \to A(\text{succ} \ x))
}{
\Gamma \vdash \text{elimNat}(x.A) \ M \ F : \Pi x : Nat (A(x))}
\]

(where $A(t)$ stands for $A[t/x]$)
Extending \( \lambda C \) with an inductive type of natural numbers

Pseudo-terms

\[
t ::= \cdots \mid \text{Nat} \mid \text{zero} \mid \text{succ} \mid \text{elimNat}(x.t) \ t 
\]

Typing rules

- formation: \( \Diamond \vdash \text{Nat} : \text{Set} \)
- introduction: \( \Diamond \vdash \text{zero} : \text{Nat} \quad \Diamond \vdash \text{succ} : \text{Nat} \rightarrow \text{Nat} \)

\[
\begin{align*}
\Gamma, x : \text{Nat} & \vdash A(x) : s \\
\Gamma & \vdash M : A(\text{zero}) \\
\Gamma & \vdash F : \Pi x : \text{Nat} (A(x) \rightarrow A(\text{succ} x)) \\
\Gamma & \vdash \text{elimNat}(x.A) \ MF : \Pi x : \text{Nat} (A(x))
\end{align*}
\]

(where \( A(t) \) stands for \( A[t/x] \))

This gives us (dep. typed) functions defined by primitive recursion, e.g.,

addition \( \lambda x : \text{Nat} (\text{elimNat}(y, \text{Nat}) \ x (\lambda y : \text{Nat} (\text{succ} y))) \)
Extending $\lambda C$ with an inductive type of natural numbers

Pseudo-terms

$$ t ::= \cdots \mid \text{Nat} \mid \text{zero} \mid \text{succ} \mid \text{elimNat}(x.t) t t $$

Typing rules

- formation: $\varnothing \vdash \text{Nat} : \text{Set}$
- introduction: $\varnothing \vdash \text{zero} : \text{Nat} \quad \varnothing \vdash \text{succ} : \text{Nat} \to \text{Nat}$

\[
\begin{align*}
\Gamma, x : \text{Nat} & \vdash A(x) : s \\
\Gamma & \vdash M : A(\text{zero}) \\
\Gamma & \vdash F : \Pi x : \text{Nat} (A(x) \to A(\text{succ} x))
\end{align*}
\]

- elimination: $\Gamma \vdash \text{elimNat}(x.A) MF : \Pi x : \text{Nat} (A(x))$

(where $A(t)$ stands for $A[t/x]$)

Computation rules

$\text{elimNat}(x.A) MF \text{zero} \to M$

$\text{elimNat}(x.A) MF (\text{succ} N) \to FN (\text{elimNat}(x.A) MF N)$
Extending $\lambda C$ with an inductive type of natural numbers

Pseudo-terms

$$t ::= \cdots \mid \text{Nat} \mid \text{zero} \mid \text{succ} \mid \text{elimNat}(x.t) t t$$

Typing rules

- formation: $\Diamond \vdash \text{Nat} : \text{Set}$
- introduction: $\Diamond \vdash \text{zero} : \text{Nat} \quad \Diamond \vdash \text{succ} : \text{Nat} \rightarrow \text{Nat}$

$$
\begin{align*}
\Gamma, x : \text{Nat} & \vdash A(x) : s \\
\Gamma & \vdash M : A(\text{zero})
\end{align*}
\quad
\begin{align*}
\Gamma, x : \text{Nat} & \vdash F : \Pi x : \text{Nat} \left( A(x) \rightarrow A(\text{succ } x) \right)
\end{align*}

\begin{align*}
\Gamma & \vdash \text{elimNat}(x.A) \quad MF : \Pi x : \text{Nat} \left( A(x) \right)
\end{align*}

(\text{where } A(t) \text{ stands for } A[t/x])

also gives us \underline{proof by induction}

$$
\varphi(\text{zero}) \land \forall x \left( \varphi(x) \rightarrow \varphi(succ x) \right)
\rightarrow \forall x \varphi(x)
$$
Inductive types of vectors

For a fixed parameter $\Gamma \vdash A : s$, the indexed family $(\text{Vec}_A x \mid x : \text{Nat})$ of types $\text{Vec}_A x$ of *lists of $A$-values of length $x$* is inductively defined as follows:
Inductive types of vectors

For a fixed parameter $\Gamma ⊢ A : s$, the indexed family $(\text{Vec}_A x \mid x : \text{Nat})$ of types $\text{Vec}_A x$ of *lists of $A$-values of length $x$* is inductively defined as follows:

**Formation:**

\[
\Gamma ⊢ N : \text{Nat} \\
\Gamma ⊢ \text{Vec}_A N : \text{Set}
\]

**Introduction:**

\[
\Gamma ⊢ \text{vnil}_A : \text{Vec}_A \text{zero}
\]

\[
\Gamma ⊢ \text{vcons}_A : A \to \Pi x : \text{Nat} (\text{Vec}_A x \to \text{Vec}_A (\text{succ } x))
\]

**Elimination and Computation:**
Inductive types of vectors

For a fixed parameter $\Gamma \vdash A : s$, the indexed family $(\text{Vec}_A x \mid x : \text{Nat})$ of types $\text{Vec}_A x$ of \textit{lists of A-values of length x} is inductively defined as follows:

Formation:

$$
\frac{\Gamma \vdash N : \text{Nat}}{\Gamma \vdash \text{Vec}_A N : \text{Set}}
$$

Introduction:

$$
\Gamma \vdash \text{vnil}_A : \text{Vec}_A \text{zero}
$$

$$
\Gamma \vdash \text{vcons}_A : A \to \prod x : \text{Nat} (\text{Vec}_A x \to \text{Vec}_A (\text{succ } x))
$$

Elimination and Computation:

[do-it-yourself]
Inductive identity propositions

For fixed parameters $\Gamma \vdash A : s$ and $\Gamma \vdash a : A$, the indexed family $(\text{Id}_{A,a} x \mid x : A)$ of propositions $\text{Id}_{A,a} x$ that *a and x are equal elements of type A* is inductively defined as follows:
Inductive identity propositions

For fixed parameters $\Gamma \vdash A : s$ and $\Gamma \vdash a : A$, the indexed family $(\text{Id}_{A,a} x \mid x : A)$ of propositions $\text{Id}_{A,a} x$ that $a$ and $x$ are equal elements of type $A$ is inductively defined as follows:

Formation:

$$
\frac{\Gamma \vdash M : A}{\Gamma \vdash \text{Id}_{A,a} M : \text{Prop}}
$$

Introduction:

$$
\Gamma \vdash \text{refl}_{A,a} : \text{Id}_{A,a} a
$$

Elimination:

$$
\frac{\Gamma, x : A, p : \text{Id}_{A,a} x \vdash B(x, p) : s \quad \Gamma \vdash N : B(a, \text{refl}_{A,a})}{\Gamma \vdash J_{A,a}(x, p.B) N : \Pi x : A \left(\Pi p : \text{Id}_{A,a} x \right) (B(x, p))}
$$

Computation:

$$
J_{A,a}(x, p.B) N \, a \, \text{refl}_{A,a} \rightarrow N
$$
Inductive identity propositions

programming/proving using eliminators gets tricky very rapidly
(cf. Ex. Sh. qu 19 about proving $\forall n (\exists r = 0 + n)$)

Elimination:

$$\Gamma, x : A, p : \text{Id}_{A,a} x \vdash B(x, p) : s \quad \Gamma \vdash N : B(a, \text{refl}_{A,a})$$
$$\Gamma \vdash J_{A,a}(x, p, B) N : \Pi x : A (\Pi p : \text{Id}_{A,a} x (B(x, p)))$$

Computation:

$$J_{A,a}(x, p, B) N a \text{refl}_{A,a} \rightarrow N$$
Agda proof of $\forall x \in \mathbb{N} \ (x = 0 + x)$

```agda
data Nat : Set where
  zero : Nat
  succ : Nat -> Nat

add : Nat -> Nat -> Nat
add x zero    = x
add x (succ y) = succ (add x y)
```
Agda proof of $\forall x \in \mathbb{N} \ (x = 0 + x)$

```agda
data Nat : Set where
  zero : Nat
  succ : Nat -> Nat

add : Nat -> Nat -> Nat
add x zero = x
add x (succ y) = succ (add x y)

data Id (A : Set)(x : A) : A -> Set where
  refl : Id A x x

cong : (A B : Set)(f : A -> B)(x y : A) ->
       Id A x y -> Id B (f x) (f y)
cong A B f x .x refl = refl
```
Agda proof of $\forall x \in \mathbb{N} \ (x = 0 + x)$

data Nat : Set where
  zero : Nat
  succ : Nat -> Nat

add : Nat -> Nat -> Nat
add x zero = x
add x (succ y) = succ (add x y)

data Id (A : Set)(x : A) : A -> Set where
  refl : Id A x x

cong : (A B : Set)(f : A -> B)(x y : A) ->
      Id A x y -> Id B (f x) (f y)
cong A B f x .x refl = refl

P : (x : Nat) -> Id Nat x (add zero x)
P zero = refl
P (succ x) = cong Nat Nat succ x (add zero x) (P x)
Uniqueness of identity proofs

In \( \lambda C \) extended with inductive identity propositions, there are some types \( \Gamma \vdash A : s \) for which it is impossible to prove that all equality proofs in \( \text{Id}_{A,x}y \) (where \( x, y : A \)) are identical. That is, there is no pseudo-term \( uip \) satisfying

\[
\Gamma \vdash uip : \Pi x, y : A \left( \Pi p, q : \text{Id}_{A,x}y \left( \text{Id}_{\text{Id}_{A,x}y,p}q \right) \right)
\]
Uniqueness of identity proofs

In $\lambda C$ extended with inductive identity propositions, there are some types $\Gamma \vdash A : s$ for which it is impossible to prove that all equality proofs in $\text{Id}_{A,x,y}$ (where $x,y : A$) are identical. That is, there is no pseudo-term $\text{uip}$ satisfying

$$\Gamma \vdash \text{uip} : \Pi x,y : A \left( \Pi p,q : \text{Id}_{A,x,y} \left( \text{Id}_{\text{Id}_{A,x,y},p,q} \right) \right)$$

By contrast, in Agda we have:

```agda
data Id (A : Set)(x : A) : A -> Set where
  refl : Id A x x

uip : (A : Set)(x y : A)(p q : Id A x y) -> Id (Id A x y) p q
uip A x .x refl refl = refl```

Dependent function types $(\Pi x : A) B$

\[
\begin{align*}
\text{ML} & \quad \text{type schemes} \\
\text{function types} & \\
\text{PLC} & \quad \text{A-types} \\
\text{function types} & \\
P\iota's & \\
F_\omega & , \lambda C \\
\Pi\iota-types & 
\end{align*}
\]
Dependent function types \((TT\alpha; A) B\)

\[
\begin{align*}
\text{PLC} & \left\{ F_\omega, \lambda C \right\} \\
\text{ML} & \quad \text{"Turing powerful" termination undecidable}
\end{align*}
\]

\text{PTT}\'s \\
only total functions: \\
termination decidable \\
\implies decidable type-checking
Dependent function types \((TT\alpha: A) B\)

\[ M\ell \]

"impure" computation has side-effects

\[ \text{PLC} \]

\[ F_\omega, \lambda C \]

pure
Dependent function types \((\mathbb{T}\alpha:A)B\)

- **ML**

- **PLC**

- **PTSS's**
  - \(F_\omega\)
  - \(\lambda C\)

"impure" computation has side-effects

- pure

- make sense of
  - Propositions-as-Types in presence of side-effects

?