Dependent types
I. What we have seen so far
A notion of proof (proof-term)

Termination proof-reduction for many theories (model-theoretic criterion: super-consistency)
But

Term, proposition, proof: distinct notions
For instance $\lambda x \ x$ or $SKK$ a function of type $\iota \to \iota$, $\lambda \alpha \alpha$ a proof of $A \Rightarrow A$
A single notion of function?

Easier to implement: only one syntactic category, substitution function, type-checking algorithm, etc.
What happens if we mix everything? More theorems?

The $\lambda$-calculus with dependent types ($\lambda\Pi$-calculus)
II. The $\lambda$-calculus with dependent types
Independently of the notions of term, proposition, proof, etc.

The size of an array of natural numbers part of its type

Not a single type *array*

A type family (*array* 0), (*array* 1), (*array* 2), etc.
$(f\ 0) = [\ ], (f\ 1) = [0], (f\ 2) = [0, 0], (f\ 3) = [0, 0, 0], \text{ etc.}$

type of the argument of $f$: $\text{nat}$

type of the result: not always the same: depends on the argument:

$(\text{array } x)$ where $x$ is the argument of the function

\[
\text{nat} \rightarrow (\text{array } x)
\]

$x$

\[
\Pi x : \text{nat} \ (\text{array } x)
\]

$A \rightarrow B$ becomes a particular case of $\Pi x : A \ B$ when $x$ not used in $B$ (needs not be indicated)
Typing types

(array 0), Πn : nat (array n) types
(array 0 0), (array true) not well-formed
Types must be typed

Context y : (array x) not well-formed (x not declared)
Type formation, context formation rules, like term formation rules
Types are terms

Types are just terms
A constant *Type* for the type of types

Judgments $\Gamma \vdash t : A$ (particular case: $\Gamma \vdash t : Type$) and $\Gamma$ well-formed
The Simply typed $\lambda$-calculus revisited

$\Gamma$ well-formed

$\Gamma, A : Type$ well-formed

$\Gamma \vdash A : Type \quad \Gamma \vdash B : Type$

$\Gamma \vdash A \rightarrow B : Type$
\[\Gamma \vdash A : \text{Type}\]
\[\Gamma, x : A \text{ well-formed}\]

\[\frac{\Gamma \text{ well-formed}}{\Gamma \vdash x : A} \quad x : A \in \Gamma\]

\[\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type} \quad \Gamma, x : A \vdash t : B\]
\[\Gamma \vdash \lambda x : A \ t : A \to B\]

\[\Gamma \vdash t : A \to B \quad \Gamma \vdash t' : A\]
\[\Gamma \vdash (t \ t') : B\]
Kinds

(array 0), (array 1) have type Type
the variable array has type nat → Type
nat → Type is a type / has type Type?

(array 0) has type Type
Type is a type / has type Type?

Type : Type → Girard’s paradox
Non terminating terms, inconsistent

A new constant Kind for the type of Type, nat → Type, ...
Four categories of terms

- **Kind**
  - kinds: $Type, nat \rightarrow Type, \ldots$ whose type is $Kind$
  - types and type families: $nat, (array\ 0), array, \ldots$ whose type is a kind
  - objects: 0, [0], \ldots whose type is a type
Terms in each category

- **Kind** only term in its category
- **Kinds**: Type and products ($\texttt{nat} \rightarrow \texttt{Type}$, that is $\Pi \! x : \! \texttt{nat} \! \texttt{Type}$)

- Types and type families: variables ($\texttt{nat}$, $\texttt{array}$), applications (($\texttt{array} \! 0$)), abstractions ($\lambda \! n : \! \texttt{nat} \! \ (\texttt{array} \! (\! S \! n))$), and products ($\Pi \! n : \! \texttt{nat} \! \ (\texttt{array} \! n)$ and $\texttt{nat} \rightarrow \texttt{nat}$, that is $\Pi \! x : \! \texttt{nat} \! \texttt{nat}$)

- **Objects**: variables ($0$), applications (($S \! 0$)) and abstractions ($\lambda \! x : \! \texttt{nat} \! \ x$)
Typing rules

\[
\begin{align*}
\Box \text{ well-formed} \\
\Gamma \vdash A : \text{Kind} \\
\Gamma, x : A \text{ well-formed} \\
\Gamma \vdash A : \text{Type} \\
\Gamma, x : A \text{ well-formed} \\
\Gamma \text{ well-formed} \\
\Gamma \vdash \text{Type} : \text{Kind} \\
\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Kind} \\
\Gamma \vdash \Pi x : A \ B : \text{Kind}
\end{align*}
\]
\[
\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type} \\
\Gamma \vdash \Pi x : A \ B : \text{Type}
\]

\[
\Gamma \text{ well-formed} \quad \frac{}{\Gamma \vdash x : A} \quad x : A \in \Gamma
\]

\[
\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Kind} \quad \Gamma, x : A \vdash t : B \\
\Gamma \vdash \lambda x : A \ t : \Pi x : A \ B
\]

\[
\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type} \quad \Gamma, x : A \vdash t : B \\
\Gamma \vdash \lambda x : A \ t : \Pi x : A \ B
\]

\[
\Gamma \vdash t : \Pi x : A \ B \quad \Gamma \vdash t' : A \\
\Gamma \vdash (t \ t') : (t'/x)B
\]
The conversion rules

\[
\frac{\Gamma \vdash t : A \quad \Gamma \vdash A : Type \quad \Gamma \vdash B : Type}{\Gamma \vdash t : B} \quad A \equiv B
\]

\[
\frac{\Gamma \vdash t : A \quad \Gamma \vdash A : Kind \quad \Gamma \vdash B : Kind}{\Gamma \vdash t : B} \quad A \equiv B
\]

A little bit of Deduction modulo theory

\[\text{array'} = \lambda n : \text{nat} \ (\text{array} \ (S \ n))\]

[0] has the type \(\text{array} \ (S \ 0)\)

and also \(\lambda n : \text{nat} \ (\text{array} \ (S \ n)) \ 0\) (i.e. \(\text{array'} \ 0\))
No polymorphism

Product rule: \( \text{nat} \rightarrow \text{Type} \)
But not \( \text{Type} \rightarrow \text{Type} \)

Arrays parametrized by the number of their elements but not by the type of their elements

An extension: the Calculus of Constructions
III. The termination of reduction in the $\lambda\Pi$-calculus
Candidates in the $\lambda\Pi$-calculus

Only one “connective”: $\Pi$

$C$ set of terms, $S$ set of sets of terms
$\tilde{\Pi}(C, S)$ set of strongly terminating terms $t$ such that if $t \longrightarrow^* \lambda x : E \ t_1$ then for all $t'$ in $C$, and for all $D$ in $S$, $(t'/x)t_1 \in D$
Candidates inductively defined by:

- the set of all strongly terminating terms in a candidate
- if $C$ is a candidate and $S$ is a set of candidates, then $\tilde{\Pi}(C, S)$ is a candidate
- if $S$ is a set of candidates, then $\bigcap S$ is a candidate
Four easy lemmas

If $C$ is a candidate, then all the elements of $C$ strongly terminate.

Let $C$ be a candidate and $x$ be a variable, then $x \in C$.

If $C$ is a candidate, $t$ is an element of $C$, and $t \rightarrow^* t'$, then $t'$ is an element of $C$.

Let $C$ be a candidate. If all the one-step reducts of the term $(u_1 \ u_2)$ are in $C$, then $(u_1 \ u_2)$ is in $C$. 
Terms of $\lambda \Pi$ are at the same time sorts, terms and propositions, and proofs $(\mathcal{M}_t)_t$ indexed by terms of $\lambda \Pi$

- if $t$ is an object or a type family, then $\mathcal{M}_t = \{e\}$
- if $t$ is a kind or $t = Kind$, then $\mathcal{M}_t = \mathcal{C}$
In a more systematic way

- $\mathcal{M}_{\text{Type}} = \mathcal{M}_{\text{Kind}} = C$
- $\mathcal{M}_x = \{e\}$, an arbitrary singleton
- $\mathcal{M}_{\lambda x:A \ t} = \mathcal{M}_t$, $\mathcal{M}(t \ u) = \mathcal{M}_t$
- $\mathcal{M}_{\Pi x:A \ B}$ is the set of functions $f$ from $\mathcal{M}_A$ to $\mathcal{M}_B$ except if $\mathcal{M}_B = \{e\}$, in which case $\mathcal{M}_{\Pi x:A \ B} = \{e\}$, or if $\mathcal{M}_A = \{e\}$, in which case $\mathcal{M}_{\Pi x:A \ B} = \mathcal{M}_B$
Let $\Gamma = x_1 : A_1, ..., x_n : A_n$ be a well-formed context. A $\Gamma$-valuation $\phi$ is a function mapping every variable $x_i$ to an element of $\mathcal{M}_{A_i}$. 
$[[t]]_{\phi}$ of $\mathcal{M}_A$ defined as follows

- $[[Type]]_{\phi}$ is the set of strongly terminating terms
- $[[Kind]]_{\phi}$ is the set of strongly terminating terms
- $[[x]]_{\phi} = \phi(x)$
- $[[\lambda x : C \ t]]_{\phi}$ is the function of domain $\mathcal{M}_C$ mapping $a$ in $\mathcal{M}_C$ to $[[t]]_{\phi,x=a}$, except if $[[t]]_{\phi,x=a} = e$ for all $a$, in which case $[[\lambda x : C \ t]]_{\phi} = e$, or if $\mathcal{M}_C = \{e\}$, in which case $[[\lambda x : C \ t]]_{\phi} = [[t]]_{\phi,x=e}$
- $[[ (t \ u) ]]_{\phi} = [[t]]_{\phi} [[u]]_{\phi}$, except if $[[t]]_{\phi} = e$, in which case $[[ (t \ u) ]]_{\phi} = e$, or if $[[u]]_{\phi} = e$, in which case $[[ (t \ u) ]]_{\phi} = [[t]]_{\phi}$
- $[[\Pi x : C \ D]]_{\phi}$ is the candidate $\tilde{\Pi}([[C]]_{\phi}, \{[[D]]_{\phi,x=c} \mid c \in \mathcal{M}_C\})$
If $t \equiv u$ then $[t]_\phi = [u]_\phi$

Let $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ be a context, $\phi$ be a $\Gamma$-valuation, $\sigma$ be a substitution mapping every $x_i$ to an element of $[A_i]_\phi$ and $t$ a term of type $B$ in $\Gamma$. Then $\sigma t \in [B]_\phi$

Let $\Gamma$ be a context and $t$ be a term well-typed in $\Gamma$. Then $t$ strongly terminates
IV. Representation of terms, propositions, and proofs of minimal logic
Minimal Logic

Fragment of Predicate logic
Only $\Rightarrow$ and $\forall$
Only rules: axiom, introduction and elimination of $\Rightarrow$ and of $\forall$
A language $\mathcal{L}$ of Predicate logic
Associate a context $\Xi$ containing

- for each sort $s$ of $\mathcal{L}$ a variable $s$ of type $\text{Type}$
- for each function symbol $f$ of arity $\langle s_1, \ldots, s_n, s' \rangle$, a variable, also written $f$, of type $s_1 \rightarrow \ldots \rightarrow s_n \rightarrow s'$
- for each predicate symbol $P$ of arity $\langle s_1, \ldots, s_n \rangle$ a variable, also written $P$, of type $s_1 \rightarrow \ldots \rightarrow s_n \rightarrow \text{Type}$
Terms and propositions

- $\Phi(x) = x$
- $\Phi(f(t_1, \ldots, t_n)) = f(\Phi(t_1), \ldots, \Phi(t_n))$

- $\Phi(P(t_1, \ldots, t_n)) = P(\Phi(t_1), \ldots, \Phi(t_n))$
- $\Phi(A \Rightarrow B) = \Phi(A) \rightarrow \Phi(B)$, that is $\Pi x : \Phi(A) \Phi(B)$
- $\Phi(\forall x \ A) = \Pi x : s \ \Phi(A)$
Proofs

A sequent $A_1, ..., A_n \vdash B$
A context $\Gamma$ containing

- $\exists$
- for each variable $x$ of sort $s$ free in $A_1, ..., A_n \vdash B$, a variable, also written $x$, of type $s$
- for each hypothesis $A_i$ a variable $\alpha_i$ of type $\Phi(A_i)$

$A_1, ..., A_n \vdash B$ has a proof iff there exists $\pi$ such that $\Gamma \vdash \pi : \Phi(B)$
A / the logical framework

Like Predicate logic
Like Deduction modulo theory
Why stick to minimal logic: the $\lambda 1$-calculus

Besides $\Pi$

Sums ($\land$, $\exists$)
Disjoint unions ($\lor$)
Unit type ($\top$)
Empty type ($\bot$)
V. The $\lambda\Pi$-calculus modulo theory
Variables $\Xi$ then reduction rules on the symbols of $\Xi$ then more variables $\Gamma$

Then replace $\equiv_\beta$ by $\equiv_{\beta R}$

$$
\frac{
\Gamma \vdash A : Type \quad \Gamma \vdash B : Type \quad \Gamma \vdash t : A
}{
\Gamma \vdash t : B}
A \equiv_{\beta R} B
$$

$$
\frac{
\Gamma \vdash A : Kind \quad \Gamma \vdash B : Kind \quad \Gamma \vdash t : A
}{
\Gamma \vdash t : B}
A \equiv_{\beta R} B
$$

Proofs of minimal Deduction modulo theory can be expressed as terms in the $\lambda\Pi$-calculus modulo theory
Minimal Simple type theory in $\lambda\Pi$-modulo theory

Drop $\top\dot{,}\bot,\land,\lor,$ and $\exists_A$ and the associated reduction rules

In minimal Deduction modulo theory, hence in $\lambda\Pi$-calculus modulo theory
But: an infinite number of sorts and symbols
Instead of: a variable for each sort
Two variables $\iota$ and $o$ of type $Type$
Translate the simple type as

- $|\iota| = \iota, |o| = o,$
- $|A \to B| = |A| \to |B|,$ that is $\Pi x : |A| |B|.$
A notation for terms based on $\lambda$-calculus and not on combinators we translate terms as

- $|x| = x,$
- $|(t\ u)| = (|t|\ |u|),$  
- $|((\lambda x : A\ t))| = \lambda x : |A|\ |t|.$

$\lambda\Pi$-calculus already contains a notion of function that may be reused instead of redefining one for Simple type theory

$\beta$-reduction of Simple type theory: $\beta$-reduction of $\lambda\Pi$-calculus
But keep $\varepsilon$, $\Rightarrow$ and $\forall_A$ and

$$
\varepsilon(\Rightarrow x y) \longrightarrow \varepsilon(x) \rightarrow \varepsilon(y)
$$

$$
\varepsilon(\forall_A x) \longrightarrow \Pi y : |A| \varepsilon(x y)
$$

Still an infinite number of symbols: can be avoided

Expression of proofs of HOL in Dedukti
Termination

- $\mathcal{M}_{\text{Type}} = \mathcal{M}_{\text{Kind}} = \mathcal{M}_o = C$
- $\mathcal{M}_l = \mathcal{M}_e = \mathcal{M}_x = \mathcal{M} \leadsto = \mathcal{M}_{\forall A} = \{ e \}$, an arbitrary singleton
- $\mathcal{M}_{\lambda x : A} t = \mathcal{M}_t$
- $\mathcal{M}(t u) = \mathcal{M}_t$
- $\mathcal{M}_{\Pi x : A} B$ is the set of functions $f$ from $\mathcal{M}_A$ to $\mathcal{M}_B$, except if $\mathcal{M}_B = \{ e \}$, in which case $\mathcal{M}_{\Pi x : A} B = \{ e \}$, or if $\mathcal{M}_A = \{ e \}$ in which case $\mathcal{M}_{\Pi x : A} B = \mathcal{M}_B$
- $[\text{Type}]_\phi$ is the set of strongly terminating terms
- $[\text{Kind}]_\phi$ is the set of strongly terminating terms
- $[\text{o}]_\phi$ is the set of strongly terminating terms
- $[\text{u}]_\phi$ is the set of strongly terminating terms
- $[x]_\phi = \phi(x)$
- $[\lambda x : C t]_\phi$ is the function of domain $\mathcal{M}_C$ mapping $a$ in $\mathcal{M}_C$ to $[t]_{\phi, x = a}$, except if $[t]_{\phi, x = a} = e$ for all $a$, in which case $[\lambda x : C t]_\phi = e$, or if $\mathcal{M}_C = \{ e \}$, in which case $[\lambda x : C t]_\phi = [t]_{\phi, x = e}$
\[\lceil(t \ u)\rceil_\phi = \lceil t\rceil_\phi(\lceil u\rceil_\phi),\] except if \(\lceil t\rceil_\phi = e\), in which case \(\lceil(t \ u)\rceil_\phi = e\), or if \(\lceil u\rceil_\phi = e\), in which case \(\lceil(t \ u)\rceil_\phi = \lceil t\rceil_\phi\).

\[\lceil \Pi x : C \ D \rceil_\phi\] is the candidate \(\hat{\Pi}(\lceil C\rceil_\phi, \{\lceil D\rceil_\phi, x = c \mid c \in \mathcal{M}_C\})\).

\[\lceil \varepsilon \rceil_\phi\] is the identity on \(C\).

\[\lceil \Rightarrow \rceil_\phi = \Rightarrow\]

\[\lceil \forall A \rceil_\phi\] is the function mapping the function \(f\) from \(\mathcal{M}_{|A|}\) to \(C\) to the candidate \(\hat{\Pi}(\lceil A\rceil_\phi, \{f(a) \mid a \in \mathcal{M}_{|A|}\})\).
Next time

Inductive types