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 $\iota \rightarrow \iota$, $\lambda \alpha \alpha$ 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], 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 \textit{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

\[ [\ ] \text{well-formed} \]

\[
\Gamma \text{ well-formed} \\
\frac{}{\Gamma, A : Type \text{ well-formed}}
\]

\[
\Gamma \vdash A : Type \quad \Gamma \vdash B : Type \\
\frac{}{\Gamma \vdash A \rightarrow B : Type}
\]
\[
\begin{align*}
\Gamma &\vdash A : Type \\
\Gamma, x : A & \text{ well-formed}
\end{align*}
\]

\[
\begin{align*}
\Gamma & \text{ well-formed} \\
\Gamma & \vdash x : A \\
\Gamma & \vdash A : Type \\
\Gamma, x : A & \vdash B : Type \\
\Gamma, x : A & \vdash t : B \\
\Gamma & \vdash \lambda x : A \ t : A \to B \\
\Gamma & \vdash t : A \to B \\
\Gamma & \vdash t' : A \\
\Gamma & \vdash (t \ t') : B
\end{align*}
\]
Kinds

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

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

Type : Type \to Girard’s paradox
Non terminating terms, inconsistent

A new constant Kind for the type of Type, nat \to 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 ($\text{nat} \rightarrow \text{Type}$, that is $\Pi x : \text{nat} \; \text{Type}$)

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

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

\[ [ ] \text{ well-formed} \]
\[ \Gamma \vdash A : Kind \]
\[ \Gamma, x : A \text{ well-formed} \]
\[ \Gamma \vdash A : Type \]
\[ \Gamma, x : A \text{ well-formed} \]
\[ \Gamma \text{ well-formed} \]
\[ \Gamma \vdash Type : Kind \]
\[ \Gamma \vdash A : Type \quad \Gamma, x : A \vdash B : Kind \]
\[ \Gamma \vdash \Pi x : A \ B : Kind \]
\[
\begin{align*}
\Gamma &\vdash A : Type & \Gamma, x : A &\vdash B : Type & \Gamma &\vdash \Pi x : A \ B : Type \\
\Gamma &\vdash \text{well-formed} & \Gamma &\vdash x : A & x : A &\in \Gamma \\
\Gamma &\vdash A : Type & \Gamma, x : A &\vdash B : \text{Kind} & \Gamma, x : A &\vdash t : B & \Gamma &\vdash \lambda x : A \ t : \Pi x : A \ B \\
\Gamma &\vdash A : Type & \Gamma, x : A &\vdash B : Type & \Gamma, x : A &\vdash t : B & \Gamma &\vdash \lambda x : A \ t : \Pi x : A \ B \\
\Gamma &\vdash t : \Pi x : A \ B & \Gamma &\vdash t' : A & \Gamma &\vdash (t \ t') : (t'/x)B
\end{align*}
\]
The conversion rules

\[
\begin{array}{c}
\frac{\Gamma \vdash t : A \quad \Gamma \vdash A : \text{Type} \quad \Gamma \vdash B : \text{Type}}{\Gamma \vdash t : B} \quad A \equiv B
\\
\frac{\Gamma \vdash t : A \quad \Gamma \vdash A : \text{Kind} \quad \Gamma \vdash B : \text{Kind}}{\Gamma \vdash t : B} \quad A \equiv B
\end{array}
\]

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: \( nat \to Type \)
But not \( Type \to Type \)

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 $t \rightarrow^* \lambda x : E \ t_1$ 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

$(M_t)_t$ indexed by terms of $\lambda \Pi$

- if $t$ is an object or a type, then $M_t = \{e\}$
- if $t$ is a kind or $t = Kind$, then $M_t = C$
Slightly more general

- $\mathcal{M}_{Type} = \mathcal{M}_{Kind} = \mathcal{C}$
- $\mathcal{M}_x = \{e\}$, 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 $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ be a well-formed context. A $\Gamma$-valuation $\phi$ is a function mapping every variable $x_i$ to an element of $M_{A_i}$.
\([ t ]_{\phi}\) of \(M_A\) defined as follows

- \(\llbracket \, Type \rrbracket_{\phi}\) is the set of strongly terminating terms
- \(\llbracket \, Kind \rrbracket_{\phi}\) is the set of strongly terminating terms
- \(\llbracket x \rrbracket_{\phi} = \phi(x)\)
- \(\llbracket \lambda x : C \ t \rrbracket_{\phi}\) is the function of domain \(M_C\) mapping \(a\) in \(M_C\) to \(\llbracket t \rrbracket_{\phi, x = a}\), except if \(\llbracket t \rrbracket_{\phi, x = a} = e\) for all \(a\), in which case \(\llbracket \lambda x : C \ t \rrbracket_{\phi} = e\), or if \(M_C = \{e\}\), in which case \(\llbracket \lambda x : C \ t \rrbracket_{\phi} = \llbracket t \rrbracket_{\phi, x = e}\)
- \(\llbracket (t \ u) \rrbracket_{\phi} = \llbracket t \rrbracket_{\phi} \llbracket u \rrbracket_{\phi}\), except if \(\llbracket t \rrbracket_{\phi} = e\), in which case \(\llbracket (t \ u) \rrbracket_{\phi} = e\), or if \(\llbracket u \rrbracket_{\phi} = e\), in which case \(\llbracket (t \ u) \rrbracket_{\phi} = \llbracket t \rrbracket_{\phi}\)
- \(\llbracket \Pi x : C \ D \rrbracket_{\phi}\) is the candidate \(\tilde{\Pi}(\llbracket C \rrbracket_{\phi}, \{\llbracket D \rrbracket_{\phi, x = c} \mid c \in M_C\})\)
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 ⇒ and ∀
Only rules: axiom, introduction and elimination of ⇒ and of ∀
Languages

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 $\Pi x : \Phi(A) \Phi(B)$
- $\Phi(\forall x \ A) = \Pi x : s \Phi(A)$
A sequent \( A_1, \ldots, A_n \vdash B \)

A context \( \Gamma \) containing

- \( \Xi \)

- for each variable \( x \) of sort \( s \) free in \( A_1, \ldots, A_n \vdash B \), a variable, also written \( x \), of type \( s \)

- for each hypothesis \( A_i \) a variable \( \alpha_i \) of type \( \Phi(A_i) \)

\( A_1, \ldots, A_n \vdash B \) 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 $\lambda I$-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 $\equiv_{\beta R}$

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

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

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 $\dot{\top}$, $\dot{\bot}$, $\dot{\land}$, $\dot{\lor}$, and $\dot{\exists}_A$ 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 \( \text{Type} \)
Translate the simple type as

\[
\begin{align*}
|\iota| &= \iota, \quad |o| = o, \\
|A \to B| &= |A| \to |B|, \text{ that is } \Pi x : |A| \to |B|.
\end{align*}
\]
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

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

$$\varepsilon(\forall_A x) \rightarrow \forall y : |A| \varepsilon(x \ y)$$

Still an infinite number of symbols: can be avoided

Expression of proofs of HOL in Dedukti
Termination

- $\mathcal{M}_{Type} = \mathcal{M}_{Kind} = \mathcal{M}_o = \mathcal{C}$
- $\mathcal{M}_t = \mathcal{M}_{\varepsilon} = \mathcal{M}_x = \mathcal{M}_{\Rightarrow} = \mathcal{M}_{\forall_A} = \{e\}$, 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 $\mathcal{M}_{\Pi x:A \ B} = \mathcal{M}_B$

- $\llbracket Type\rrbracket_\phi$ is the set of strongly terminating terms
- $\llbracket Kind\rrbracket_\phi$ is the set of strongly terminating terms
- $\llbracket \sigma \rrbracket_\phi$ is the set of strongly terminating terms
- $\llbracket \iota \rrbracket_\phi$ is the set of strongly terminating terms
- $\llbracket x \rrbracket_\phi = \phi(x)$
- $\llbracket \lambda x : C \ t \rrbracket_\phi$ is the function of domain $\mathcal{M}_C$ mapping $a$ in $\mathcal{M}_C$ to $\llbracket t \rrbracket_{\phi,x=a}$, except if $\llbracket t \rrbracket_{\phi,x=a} = e$ for all $a$, in which case $\llbracket \lambda x : C \ t \rrbracket_\phi = e$, or if $\mathcal{M}_C = \{e\}$, in which case $\llbracket \lambda x : C \ t \rrbracket_\phi = \llbracket t \rrbracket_{\phi,x=e}$
\begin{itemize}
  \item $\llbracket (t \ u) \rrbracket_\phi = \llbracket t \rrbracket_\phi (\llbracket u \rrbracket_\phi)$, except if $\llbracket t \rrbracket_\phi = e$, in which case $\llbracket (t \ u) \rrbracket_\phi = e$, or if $\llbracket u \rrbracket_\phi = e$, in which case $\llbracket (t \ u) \rrbracket_\phi = \llbracket t \rrbracket_\phi$,
  \item $\llbracket \Pi x : C \ D \rrbracket_\phi$ is the candidate $\tilde{\Pi}(\llbracket C \rrbracket_\phi, \{ \llbracket D \rrbracket_{\phi, x = c} \mid c \in \mathcal{M}_C \})$
  \item $\llbracket \varepsilon \rrbracket_\phi$ is the identity on $C$
  \item $\llbracket \Rightarrow \rrbracket_\phi = \Rightarrow$
  \item $\llbracket \forall A \rrbracket_\phi$ is the function mapping the function $f$ from $\mathcal{M}_{|A|}$ to $C$ to the candidate $\tilde{\Pi}(\llbracket A \rrbracket_\phi, \{ f(a) \mid a \in \mathcal{M}_{|A|} \})$
\end{itemize}
Next time

Inductive types