Calculemus Autumn School
Pisa September 2002

Type Theory for Formalizing Maths

Herman Geuvers
Nijmegen University, NL

Introduction to Type Theory
(Especially related to the type theory of Coq, but more widely applicable.)

- A data type (or set) is a term A : Set
- A formula is a term φ : Prop
- An object is a term $t : A$ for some A : Set
- A proof is a term $p : \varphi$ for some φ : Prop.
- Set and Prop are both of type Type.

Slogan: (Curry-Howard isomorphism)

Propositions as Types
Proofs as Terms

Judgement

$$\Gamma \vdash M : U$$

- Γ is a context
- M is a term
- U is a type

Two readings

- M is an object (expression) of data type U (if U : Set)
- M is a proof (deduction) of proposition U (if U : Prop)

Γ contains

- variable declarations $x : T$
  - $x : A$ with $A :$ Set $\rightsquigarrow$ 'declaring x in A'
  - $x : \varphi$ with $\varphi :$ Prop $\rightsquigarrow$ 'assuming φ' (axiom)
- definitions $x := M : T$
  - $x := t : A$ with $A :$ Set $\rightsquigarrow$ 'defining x as the expression t'
  - $x := p : \varphi$ with $\varphi :$ Prop $\rightsquigarrow$ 'defining x as the proof p of φ'
    ($\simeq$ declaring x as a "reference" to the lemma φ)

Dependent Function types

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}\,(\lambda) \qquad \frac{\Gamma \vdash F : \Pi x{:}A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash F\,N : B[N/x]}\,(\mathsf{app})$$

- $\Pi x{:}A.B$ is the type of functions f that take $a : A$ to $B[a/x]$.
  Coq notation: `(x:A)B`
- If $x \notin \mathsf{FV}(B)$, we write $A {\rightarrow} B$.
  Coq notation: `A -> B`
- There is a side condition on the (λ)-rule: $\Pi x{:}A.B$ must be well-formed
  We require $\Pi x{:}A.B : s$ for s a 'sort' (universe)

Formation of Π-types

- For instance:

$$\frac{\Gamma \vdash A : \mathsf{Set} \quad \Gamma, x{:}A \vdash B : \mathsf{Prop}}{\Gamma \vdash \Pi x{:}A.B : \mathsf{Prop}} \qquad \frac{\Gamma \vdash A : \mathsf{Set} \quad \Gamma, x{:}A \vdash B : \mathsf{Set}}{\Gamma \vdash \Pi x{:}A.B : \mathsf{Set}}$$

read as: $\forall x{:}A.B$ read as: $\{f \mid \forall a{:}A(f\,a \in B[a/x])\}$

Not all 'combinations' are allowed (for inconsistency reasons), but we do have:

- The combination (Set,Set) forms the function types $A{\to}B$ for A, B:Set, also type-dependent.
- (Prop,Prop) forms the propositional types $\varphi{\to}\psi$ for φ, ψ:Prop; implicational formulas.
- The combination (Set,Type) forms the unary predicate types and binary relations types: $A{\to}$Prop and $A{\to}A{\to}$Prop for A : Set.

- The combination (Type,Type) forms higher order predicate types like $(A \to A \to \mathsf{Prop}) \to \mathsf{Prop}$.
- (Type,Prop) forms the dependent propositional type $\Pi x{:}A.\varphi$ for $A{:}\mathsf{Type}$, $\varphi{:}\mathsf{Prop}$, e.g. $\Pi P{:}\mathsf{nat} \to \mathsf{Prop}.\varphi$, h.o. universally quantified formulas.

Example of the Curry-Howard isomorphism:
Deriving irreflexivity from anti-symmetry

$$
\begin{aligned}
\mathsf{Rel} &:= \lambda X{:}\mathsf{Set}.X \to X \to \mathsf{Prop}\\
\mathsf{AntiSym} &:= \lambda X{:}\mathsf{Set}.\lambda R{:}(\mathsf{Rel}\, X).\forall x, y{:}X.(Rxy) \to (Ryx) \to \bot\\
\mathsf{Irrefl} &:= \lambda X{:}\mathsf{Set}.\lambda R{:}(\mathsf{Rel}\, X).\forall x{:}X.(Rxx) \to \bot
\end{aligned}
$$

Derivation in higher order logic:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x^A y^A R\,x\,y \to R\,y\,x \to \bot}{\forall y^A R\,x\,y \to R\,y\,x \to \bot}}{R\,x\,x \to R\,x\,x \to \bot} \qquad [R\,x\,x]}{R\,x\,x \to \bot} \qquad [R\,x\,x]}{\bot}}{R\,x\,x \to \bot}}{\forall x^A.R\,x\,x \to \bot}
$$

Derivation in higher order logic, with terms:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{z : \forall x^A y^A R\,x\,y \to R\,y\,x \to \bot}{zx : \forall y^A R\,x\,y \to R\,y\,x \to \bot}}{zxx : R\,x\,x \to R\,x\,x \to \bot} \qquad [q : R\,x\,x]}{zxxq : R\,x\,x \to \bot} \qquad [q : R\,x\,x]}{zxxqq : \bot}}{\lambda q{:}(R\,x\,x).zxxqq : R\,x\,x \to \bot}}{\lambda x{:}A.\lambda q{:}(R\,x\,x).zxxqq : \forall x^A.R\,x\,x \to \bot}
$$

The associated typing judgement, following the Curry-Howard isomorphism is:

$$
A{:}\mathsf{Set}, R{:}A{\to}A{\to}\mathsf{Prop},\ z : \Pi x, y{:}A.(R\,x\,y{\to}R\,y\,x{\to}\bot) \vdash
$$
$$
\lambda x{:}A.\lambda q{:}(R\,x\,x).z\,x\,x\,q\,q : (\Pi x{:}A.R\,x\,x{\to}\bot)
$$

Type theory as a basis for theorem proving

- Interactive theorem proving = interactive term construction
  Proving φ = (interactively) constructing a *proof term* $p : \varphi$
- Proof checking = Type checking
  Type checking is decidable and hence proof checking is.

NB Proof terms are first class citizens.

Type theory as a basis for theorem proving

- Interactive theorem proving $=$ interactive term construction
  Proving $\varphi =$ (interactively) constructing a *proof term* $p : \varphi$
- Proof checking $=$ Type checking
  Type checking is decidable and hence proof checking is.

Decidability problems:

$\Gamma \vdash M : A?$	Type Checking Problem	TCP
$\Gamma \vdash M : ?$	Type Synthesis Problem	TSP
$\Gamma \vdash ? : A$	Type Inhabitation Problem	TIP

TCP and TSP are decidable
TIP is undecidable

De Bruijn criterion for theorem provers / proof checkers:
How to check the checker?

Interactive Theorem Prover:

USER $\rightleftarrows$ $\boxed{\text{Proof Engine, big, with lots of automation, generating (potential?) proofs}}$ $\xrightarrow{\text{proof term}}$ $\boxed{\text{Proof Checker, small program, user verifiable}}$ $\downarrow$ OK (Checked!)

A TP satisfies the De Bruijn criterion if a small, 'easily' verifiable, independent proof checker can be written.

How proof terms occur (in Coq):

```
Lemma trivial : (x:A)(P x) -> (P x).
Intros x H.
Exact H.
Qed.
```

- Using the tactic script a term of type `(x:A)(P x) -> (P x)` has been created.
- Using `Qed`, `trivial` is defined as this term and added to the global context.

Computation

- (β):
$$(\lambda x{:}A.M)N \to_\beta M[N/x]$$
- (ι): primitive recursion reduction rules (later)
- (δ): definition unfolding: if $x := t : A \in \Gamma$, then
$$M(x) \to_\delta M(t)$$
- Transitive, reflexive, symmetric closure: $=_{\beta\iota\delta}$

NB: Types that are equal modulo $=_{\beta\iota\delta}$ have the same inhabitants (definitional equality):

$$\text{(conversion)}\ \frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B}\ A =_{\beta\iota\delta} B$$

This is also called the Poincaré principle: "(computational) equalities do not require a proof"

Data types and executable programs in type theory

Data types:

```
Inductive nat : Set :=
    O : nat
  | S : nat->nat.
```

This definition yields

- The constructors 0 and S
- Induction principle:
  `nat_ind` : $\forall P{:}\mathsf{nat}{\to}\mathsf{Prop}.(P\,0) \to (\forall n{:}\mathsf{nat}.(P\,n){\to}(P(S\,n))) \to \forall n{:}\mathsf{nat}(P\,n)$
- Recursion scheme (primitive recursion over higher types)

Example of the recursion scheme (`(1)` abbreviates `(S O)` etc.)

```
Fixpoint nfib [n:nat] : nat :=
Cases n of
 | O     => (1)
 | (S m) => (Cases m of
             | O     => (1)
             | (S p) => (plus (plus (nfib p)(nfib m)) (1))
            end)
end.
```

NB: Recursive calls should be 'smaller' (according to some rather general syntactic measure)

- Coq includes a (small, functional) programming language in which executable functions can be written.

Dependently typed data types: vectors of length n over A

```
Inductive vect [A:Set] : nat -> Set :=
    nnil  : (vect A O)
  | ccons : (a:A;n:nat)(vect A n)->(vect A (S n)).
```

Now define, for example,

- head : ΠA:Set.Πn:nat.(vect $A(S\,n)) \to A$
- tail : ΠA:Set.Πn:nat.(vect $A(S\,n)) \to$ (vect $A\,n$)

Implicit Syntax

If the type checker can infer some arguments, we can leave them away:

Write `(f ? a b)` in stead of `(f S a b)` if `f : (S:Set)S->S->S`

Also: define `F:=(f ?)` and write `(F a b)`.

Inductive types are also used to define the logical connectives:

```
Inductive or [A : Prop; B : Prop]  : Prop :=
     or_introl : A->A\/B | or_intror : B->A\/B.

Inductive and [A : Prop; B : Prop]  : Prop :=
     conj : A->B->A/\B.

Inductive ex [A : Set; P : A->Prop]  : Prop :=
     ex_intro : (x:A)(P x)->(Ex P).

Inductive True  : Prop :=  I : True.

Inductive False  : Prop :=  .
```

All (constructive) logical rules are now derivable.

Proof terms in intensional type theory

- The 'subtype' $\{t : A \mid (P\ t)\}$ is defined as the type of pairs $\langle t, p\rangle$ where $t : A$ and $p : (P\ t)$.
  Notation: $\Sigma x{:}A.P\ x$

- A partial function is a function on a subtype
  E.g. $(-)^{-1} : \Sigma x{:}\mathbb{R}.x \neq 0 \to \mathbb{R}$.
  If $x : \mathbb{R}$ and $p : x \neq 0$, then $\frac{1}{\langle x,p\rangle} : \mathbb{R}$.

- A partial function must be proof-irrelevant, i.e.
  if $p : t \neq 0$ and $q : t \neq 0$, then $\frac{1}{\langle t,p\rangle} = \frac{1}{\langle t,q\rangle}$.

Use Σ-types for mathematical structures:
theory of groups: Given A : Type, a group over A is a tuple consisting of

$$\begin{aligned} \circ &: A \to A \to A \\ e &: A \\ \mathsf{inv} &: A \to A \end{aligned}$$

such that the following types are inhabited.

$$\begin{aligned} \Pi x, y, z{:}A.(x \circ y) \circ z &= x \circ (y \circ z), \\ \Pi x{:}A.e \circ x &= x, \\ \Pi x{:}A.(\mathsf{inv}\ x) \circ x &= e. \end{aligned}$$

Type of group-structures over A, Group-Str(A), is

$$(A \to A \to A) \times (A \times (A \to A))$$

The type of groups over A, $\mathsf{Group}(A)$, is

$$\begin{aligned}\mathsf{Group}(A) := \Sigma \circ{:}A{\to}A{\to}A.\Sigma e{:}A.\Sigma \mathsf{inv}{:}A{\to}A.\\ (\Pi x, y, z{:}A.(x \circ y) \circ z = x \circ (y \circ z)) \wedge \\ (\Pi x{:}A.e \circ x = x) \wedge \\ (\Pi x{:}A.(\mathsf{inv}\ x) \circ x = e).\end{aligned}$$

If t : $\mathsf{Group}(A)$, we can extract the elements of the group structure by projections: $\pi_1 t : A{\to}A{\to}A$, $\pi_1(\pi_2 t) : A$
If $f : A{\to}A{\to}A$, $a : A$ and $h : A{\to}A$ with p_1, p_2 and p_3 proof-terms of the associated group-axioms, then

$$\langle f, \langle a, \langle h, \langle p_1, \langle p_2, p_3\rangle\rangle\rangle\rangle\rangle : \mathsf{Group}(A).$$

We would like to use names for the projections:
Coq has labelled record types (type dependent)

- ```
  Record My_type : Set :=
    { l_1   : type_1 ;
      l_2   : type_2 ;
      l_3   : type_3   }.
  ```
  If `X : My_type`, then `(l_1 X) : type_1`.

- Also with dependent types: `l_1` may occur in `type_2`.
  If `X : My_type`, then

  `(l_2 X) : type_2 [(l_1 X)/l_1]`
  ```

- ```
  Record Group : Type :=
    { crr   :  Set;
      op    :  crr -> crr -> crr;
      unit  :  crr;
      inv   :  crr -> crr;
      assoc :  (x,y,z:crr)
               (op (op x y) z) = (op x (op y z))
      ...      ...
     }.
  ```

  If `X : Group`, then `(op X) : (crr X) -> (crr X) -> (crr X)`.

The record types can be defined in Coq using inductive types.
Note: `Group` is in `Type` and not in `Set`
  ```

Coercions

- Coercions: The user can tell the type checker to use specific terms as coercions.
  `Coercion k : A >-> B` declares the term `k : A -> B` as a coercion.
  - If `f a` can not be typed, the type checker will try to type check `(k f) a` and `f (k a)`.
  - If we declare a variable `x:A` and `A` is not a type, the type checker will check if `(k A)` is a type.

  Coercions can be composed.

Coercions and structures

```
Record CMonoid : Type :=
  { m_crr    :> CSemi_grp;
    m_proof :  (Commutative m_crr (sg_op m_crr))
       /\ (IsUnit m_crr (sg_unit m_crr) (sg_op m_crr))
  }.
```

- A monoid is now a tuple $\langle\langle\langle S, =_S, r\rangle, a, f, p\rangle, q\rangle$

  If `M : Monoid`, the carrier of `M` is `(crr(sg_crr(m_crr M)))`

  Nasty !!

  $\Rightarrow$ We want to use the structure `M` as synonym for the carrier set `(crr(sg_crr(m_crr M)))`.

  $\Rightarrow$ The maps `crr`, `sg_crr`, `m_crr` should be left implicit.

- The notation `m_crr :> Semi_grp` declares the coercion `m_crr : Monoid >-> Semi_grp`.

Functions and Algorithms

- Set theory (and logic): a function $f : A \to B$ is a relation $R \subset A \times B$ such that $\forall x{:}A.\exists! y{:}B.R\,x\,y$.
  "functions as graphs"
- In Type theory, we have functions-as-graphs ($R : A \to B \to \mathsf{Prop}$), but also functions-as-algorithms: $f : A \to B$.

Functions as algorithms also compute: β and ι rules:

$$
\begin{aligned}
(\lambda x{:}A.M)N &\longrightarrow_\beta M[N/x],\\
\mathsf{Rec}\,b\,f\,0 &\longrightarrow_\iota b,\\
\mathsf{Rec}\,b\,f\,(S\,x) &\longrightarrow_\iota f\,x\,(\mathsf{Rec}\,b\,f\,x).
\end{aligned}
$$

Terms of type $A \to B$ denote algorithms, whose operational semantics is given by the reduction rules.
(Type theory as a small programming language)

Intensionality versus Extensionality

The equality in the side condition in the (conversion) rule can be intensional or extensional.

Extensional equality requires the following rules:

$$(\text{ext}) \quad \frac{\Gamma \vdash M, N : A{\to}B \quad \Gamma \vdash p : \Pi x{:}A.(Mx = Nx)}{\Gamma \vdash M = N : A{\to}B}$$

$$(\text{conv}) \quad \frac{\Gamma \vdash P : A \quad \Gamma \vdash A = B : s}{\Gamma \vdash P : B}$$

- Intensional equality of functions = equality of algorithms (the way the function is presented to us (syntax))
- Extensional equality of functions = equality of graphs (the (set-theoretic) meaning of the function (semantics))

Adding the rule (ext) renders TCP undecidable:

Suppose $H : (A \rightarrow B) \rightarrow \mathrm{Prop}$ and $x : (H\ f)$; then

$$x : (H\ g) \text{ iff there is a } p : \Pi x{:}A.f\ x = g\ x$$

So, to solve this TCP, we need to solve a TIP.

The interactive theorem prover Nuprl is based on extensional type theory.

Poincaré Principle

> "An equality involving a computation does not require a proof"

In type theory: if $t = q$ by evaluation (computing an algorithm), then this is a trivial equality, proved by reflexivity.
This is made precise by the conversion rule:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : B}\ A =_{\beta\iota\delta} B$$

Can we actually use the programming power of Type Theory when formalizing mathematics?

Yes. For automation: replacing a proof obligation by a computation

Reflection Suppose

- We have a class of problems with a syntactic encoding as a data type, say via the type Problem.
  Example: equalities between expressions over a group
  Then the syntactic encoding is

```
Inductive E : Set :=
    evar  : nat -> E
  | eone  : E
  | eop   : E -> E -> E
  | einv  : E -> E
```

- We have a decoding function $[\![-]\!] : \text{Problem} \to \text{Prop}$
- We have a decision function $\text{Dec} : \text{Problem} \to \{0, 1\}$
- We can prove $\text{Ok} : \forall p{:}\text{Problem}((\text{Dec}(p) = 1) \to [\![p]\!])$

To verify P (from the class of problems):

- Find a p : Problem such that $[\![p]\!] = P$.
- Then $\mathsf{Dec}(p)$ yields either 1 or 0
- If $\mathsf{Dec}(p) = 1$, then we have a proof of P (using Ok)
- If $\mathsf{Dec}(p) = 0$, we obtain no information about P (it 'fails')

Note: if Dec is complete:

$$\forall p{:}\mathsf{Problem}((\mathsf{Dec}(p) = 1) \leftrightarrow [\![p]\!])$$

then $\mathsf{Dec}(p) = 0$ yields a proof of $\neg P$.

Setoids

How to represent the notion of set?

Note: A set is not just a type, because

$M : A$ is decidable whereas $t \in X$ is undecidable

A setoid is a pair $[A, =]$ with

- A : Set,
- $= \, : A \to (A \to \mathrm{Prop})$ an equivalence relation over A

Function space setoid (the setoid of setoid functions)

$[A \xrightarrow{s} B, =_{A \xrightarrow{s} B}]$ is defined by

$$A \xrightarrow{s} B \; := \; \Sigma f{:}A \to B.(\Pi x, y{:}A.(x =_A y) \to ((f\ x) =_B\ (f\ y))),$$

$$f =_{A \xrightarrow{s} B} g \; := \; \Pi x, y{:}A.(x =_A y) \to (\pi_1\ f\ x) =_B (\pi_1\ g\ y).$$

Two mathematical constructions: quotient and subset for setoids.

Q is an equivalence relation over the setoid $[A, =_A]$ if

- $Q : A {\rightarrow} (A {\rightarrow} \mathsf{Prop})$ is an equivalence relation,
- $=_A \subset Q$, i.e. $\forall x, y{:}A. (x =_A y) {\rightarrow} (Q\ x\ y)$.

The quotient setoid $[A, =_A]/Q$ is defined as

$$[A, Q]$$

Easy exercise:
If the setoid function $f : [A, =_A] \to [B, =_B]$ respects Q
(i.e. $\forall x, y{:}A. (Q\ x\ y) {\rightarrow} ((f\ x) =_B (f\ y))$)
it induces a setoid function from $[A, =_A]/Q$ to $[B, =_B]$.

Given $[A, =_A]$ and predicate P on A define the sub-setoid

$$[A, =_A] \mid P \; := \; [\Sigma x{:}A.(P\ x), =_A|P]$$

$=_A|P$ is $=_A$ restricted to P: for $q, r : \Sigma x{:}A.(P\ x)$,

$$q\ (=_A|P)\ r \; := \; (\pi_1\ q) =_A (\pi_1\ r)$$

Proof-irrelevance is "embedded" in the subsetoid construction:

Setoid functions are proof-irrelevant.