Types Summer School
Gothenburg Sweden August 2005

Formalising Mathematics in Type Theory

Herman Geuvers
Radboud University Nijmegen, NL

Dogma of Type Theory

- Everything has a type

$$M:A$$

- Types are a bit like sets, but: ...
  - types give "syntactic information"

$$3 + (7 * 8)^5 : \mathsf{nat}$$

  - sets give "semantic information"

$$3 \in \{n \in \mathbb{N} \mid \forall x, y, z > 0(x^n + y^n \neq z^n)\}$$

Per Martin-Löf:
A type comes with
construction principles: how to build objects of that type? and
elimination principles: what can you do with an object of that type?

This fits well with the Brouwerian view of mathematics:

"there exists an x" means
"we have a method of constructing x"

In short: a type is characterised by the construction principles for its objects.

Examples

- A summer school is constructed from students, teachers, a team of good organisers and good weather.
- A phrase is constructed from a noun and a verb or from two phrases with the word "and" between them.
  So any phrase has the shape
  "noun verb and noun verb and ... and noun and verb".
- A natural number is either 0 or the successor S applied to a natural number.
  So the natural numbers are the objects of the shape $S(\ldots S(0)\ldots)$.

Note:

Checking whether an object belongs to an alleged type is decidable!

But if type checking should be decidable, there is not much information one can encode in a type (?)

$$X := \{n \in \mathbb{N} \mid \forall x, y, z > 0(x^n + y^n \neq z^n)\}$$

is X a type?
The proper question is: what are the objects of X? (How does one construct them?)

One constructs an object of the type X by giving an $N \in \mathbb{N}$ and a proof of the fact that $\forall x, y, z > 0(x^N + y^N \neq z^N)$.

The type X consists of pairs $\langle N, p\rangle$, with

- $N \in \mathbb{N}$
- p a proof of $\forall x, y, z > 0(x^N + y^N \neq z^N)$

$\langle N, p\rangle : X$ is decidable (if proof-checking is decidable).

More technically.
(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 "universes" or "sorts".

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 φ : 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 φ : Prop $\rightsquigarrow$ 'defining x as the proof p of φ'
    ($\simeq$ declaring x as a "reference" to the lemma φ)

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

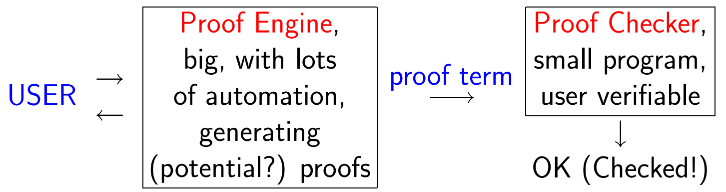
Decidability problems:

$\Gamma \vdash M : A?$	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:



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 : forall x:A, P x -> P x.
intros x H.
exact H.
Qed.
```

- Using the tactic script a term of type `forall 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"

The Poincaré principle says that if $x : A(n) \to B$ and $y : A(f\,m)$, then

$$x\,y : B \text{ iff } f\,m = n.$$

But: type checking should be decidable, so $f\,m = n$ should be decidable.

So: the definable functions in our type theory must be restricted: all computations should terminate.

Data types and executable programs in type theory

Data types:

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

This definition yields

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

Example of the recursion scheme (`1` abbreviates `(S 0)` etc.)

```
Fixpoint nfib (n:nat) :nat :=
match n with
 | 0     => 1
 | S m => match m with
             | 0     => 1
             | S p  =>  nfib p + nfib m
            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 0
    | ccons : forall (n:nat)(a:A), vect A n -> vect A (S n).
```

Now define, for example,

- head : forall (A:Set)(n:nat), vect A (S n) $\rightarrow$ A
- tail : forall (A:Set)(n:nat), vect A (S n) $\rightarrow$ vect A n

...

Let the type checker do the work for you!
Implicit Syntax
If the type checker can infer some arguments, we can leave them out:

Write $f\,\_\,\_\,a\,b$ in stead of $f\,S\,T\,a\,b$ if
$f : \Pi S, T{:}\mathsf{Set}.S \rightarrow T \rightarrow T$

Also: define $F := f\,\_\,\_$ and write $F\,a\,b$.
...

Inductive types are also used to define the logical connectives:
(Notation: `A\/B` denotes `or A B` etc.)
Inductive or (A : Prop)(B : Prop) : Prop :=
`or_introl` : A $\rightarrow$ A\/B |
`or_intror` : B $\rightarrow$ A\/B.
Inductive and (A : Prop)(B : Prop) : Prop :=
`conj` : A $\rightarrow$ B $\rightarrow$ A/\B.

Inductive ex (A : Set)(P : A$\rightarrow$Prop) : Prop :=
`ex_intro` : (x:A)(P x) $\rightarrow$ (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 $p : (P\ t)$.
- A partial function is a function on a subtype
  E.g. $(-)^{-1} : \{x{:}\mathbb{R} \mid x \neq 0\} \rightarrow \mathbb{R}$.
  If $x : \mathbb{R}$ and $p : x \neq 0$, then $\frac{1}{\langle x,p\rangle} : \mathbb{R}$.
- Usually we only consider partial functions that are 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.

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

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, Group(A), is

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

If t : 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`.
- Basically, `My_type` consists of labelled tuples:
  `[l_1:= value_1, l_2:=value_2, l_3:=value_3]`
- 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`
  ```

Let the checker infer even more for you! 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 !!
  ⇒ We want to use the structure `M` as synonym for the carrier set `(crr(sg_crr(m_crr M)))`.
  ⇒ 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 ($R : A{\to}B{\to}$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:

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

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

- 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 $H : (A \to B) \to \mathrm{Prop}$ 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.

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

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

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. $\forall x, y{:}A.(Q\ x\ y) \to ((f\ x) =_B (f\ y))$)
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.