Proofs as Programs Summer School
Eugene Oregon June - July 2002

Type Systems

Herman Geuvers
Nijmegen University, NL

Lecture 6: Various Type Theoretic Topics

Pure Type Systems

Determined by a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ with

- $\mathcal{S}$ the set of sorts
- $\mathcal{A}$ the set of axioms, $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$
- $\mathcal{R}$ the set of rules, $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$

If $s_2 = s_3$ in $(s_1, s_2, s_3) \in \mathcal{R}$, we write $(s_1, s_2) \in \mathcal{R}$.

pseudoterms:

$$\mathsf{T} ::= \mathcal{S} \mid \mathsf{Var} \mid (\Pi \mathsf{Var}{:}\mathsf{T}.\mathsf{T}) \mid (\lambda \mathsf{Var}{:}\mathsf{T}.\mathsf{T}) \mid \mathsf{TT}.$$

$$(\mathsf{sort}) \quad \vdash s_1 : s_2 \quad \text{if } (s_1, s_2) \in \mathcal{A} \qquad (\mathsf{var}) \quad \frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A} \quad \text{if } x \notin \Gamma$$

$$(\mathsf{weak}) \quad \frac{\Gamma \vdash A : s \qquad \Gamma \vdash M : C}{\Gamma, x{:}A \vdash M : C} \quad \text{if } x \notin \Gamma$$

$$(\Pi) \quad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_3} \quad \text{if } (s_1, s_2, s_3) \in \mathcal{R}$$

$$(\lambda) \quad \frac{\Gamma, x{:}A \vdash M : B \qquad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}$$

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

$$(\mathsf{conv}_\beta) \quad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B} \quad A =_\beta B$$

Examples of PTSs

CC	
$\mathcal{S}$	Prop, Type
$\mathcal{A}$	Prop : Type
$\mathcal{R}$	(Prop, Prop), (Prop, Type), (Type, Prop), (Type, Type)

CC^∞	
$\mathcal{S}$	Prop, $\{\text{Type}_i\}_{i\in\mathbb{N}}$
$\mathcal{A}$	Prop : Type, Type_i : Type_{i+1}
$\mathcal{R}$	(Prop, Prop), (Prop, Type_i), (Type_i, Prop)
	(Type_i, Type_j, $\text{Type}_{\max(i,j)}$)

Recall that ($\text{Type}_1 \text{Type}_0$, Type_0) (and similarly ($\text{Type}_{i+1} \text{Type}_i$, Type_i)) would be inconsistent.

The Extended Calculus of Constructions has in additio

- Cumulativity: Prop $\subseteq$ Type$_0$ $\subseteq$ Type$_1$ $\subseteq \ldots$, so

$$\frac{\Gamma \vdash A : \mathsf{Prop}}{\Gamma \vdash A : \mathsf{Type}_0} \quad \frac{\Gamma \vdash A : \mathsf{Type}_i}{\Gamma \vdash A : \mathsf{Type}_{i+1}}$$

- Σ-types:

$$\frac{\Gamma \vdash A : \mathsf{Prop} \quad \Gamma, x{:}A \vdash B : \mathsf{Prop}}{\Gamma \vdash \Sigma x{:}A.B : \mathsf{Prop}} \quad \frac{\Gamma \vdash A : \mathsf{Type}_i \quad \Gamma, x{:}A \vdash B : \mathsf{Type}_j}{\Gamma \vdash \Sigma x{:}A.B : \mathsf{Type}_{\max(i,j)}}$$

For φ : Prop

- We have $\Pi A{:}\mathsf{Type}_i.\varphi$: Prop, but
- We do not have $\Sigma A{:}\mathsf{Type}_i.\varphi$: Prop.

Note: The type theory of Coq has in addition Set : Type and rules $(\mathsf{Set}, \mathsf{Set}), (\mathsf{Type}_i, \mathsf{Set}), (\mathsf{Set}, \mathsf{Prop})$.

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, 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 : 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, p_3 and p_4 proof-terms of the associated group-axioms, then

$$\langle f, \langle a, \langle h, \langle p_1, \langle p_2, \langle p_3, p_4\rangle\rangle\rangle\rangle\rangle\rangle : \mathsf{Group}(A).$$

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`
  ```

Allowing

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma, x{:}A \vdash B : \mathsf{Prop}}{\Gamma \vdash \Sigma x{:}A.B : \mathsf{Prop}}$$

leads to inconsistency:

- Define $\Omega := \Sigma A{:}\mathsf{Set}.\Sigma R{:}A{\to}A{\to}\mathsf{Prop}.\mathsf{wf}(R)$
  $\mathsf{wf}(R)$ denotes that R is well-founded.
- Define $<$ on Ω by

  $(A, R) < (B, Q) :=$ R can be embedded into Q under some $b : B$

  Then

  – $<$ is well-founded on Ω

  – If (A, R) well-founded, then $(A, R) < (\Omega, <)$

  so contradiction: $\ldots < (\Omega, <) < (\Omega, <) < (\Omega, <)$.

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, 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 can be seen as a small programming language esp. if we have inductive types (with primitive recursion)

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:

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

Can we actually use the programming power of CIC 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 $[\![-]\!]$: Problem $\rightarrow$ Prop
- We have a decision function Dec : Problem $\rightarrow \{0, 1\}$
- We can prove Ok : $\forall p$:Problem$((\mathsf{Dec}(p) = 1) \rightarrow [\![p]\!])$

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 $\neg P$.

Implicit Syntax and Coercions

- 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)`.

- 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`.

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

Function space setoid

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

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

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

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 $f : [A, =_A] \rightarrow [B, =_B]$ 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

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

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

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

NB We do not require the predicate P to respect $=_A$: In taking a subsetoid we may remove elements from the $=$-classes.

Example (rational numbers): Let $A :=$ int $\times$ nat

We define, for $\langle x,p\rangle, \langle y,q\rangle{:}A$,

$$\langle x,p\rangle =_A \langle y,q\rangle \;\; := \;\; x(q+1) = y(p+1).$$

Take the predicate P on A defined by

$$Q\ \langle x,p\rangle \;\; := \;\; \gcd(x,p+1) = 1.$$

- The subsetoid $[A,=_A]|P$ is isomorphic to $[A,=_A]$
- All equivalence classes are reduced to a one element set

Objects depending on proofs

What should the type of the reciprocal?
given that the reciprocal of 0 is not defined.

- Let recip : $A \to A$ with the property $\forall x{:}A.x \neq 0 \to \mathrm{mult}x(\mathrm{recip}x) = 1$
- Now recip0 is an 'unspecified' element of A, it should be undefined
- Type theoretic solution

$$\mathrm{recip} : (\Sigma x{:}A.x \neq 0) \to A.$$

- Then recip is only defined on the subset of elements that are non-zero:
  recip takes as input a pair $\langle a, p\rangle$ with $p : a \neq 0$ and returns recip$\langle a, p\rangle : A$.

- How to understand the dependency of this object (of type A) on the proof p?

Possible solution: setoids

- Take a setoid $[A, =_A]$ as the carrier of a field
- The operations on the field are taken to be setoid functions
- The field-properties are now denoted using the setoid equality. For the reciprocal:

$$\text{recip} : [A, =_A] | (\lambda x{:}A.x \neq_A 0) \to [A, =_A],$$

a setoid function from the subsetoid of non-zeros to $[A, =_A]$

Note recip still takes a pair of an object and a proof $\langle a, p \rangle$ and returns $\text{recip}\langle a, p \rangle : A$.
But recip is now a setoid function which implies

If $p : a \neq_A 0, q : a \neq_A 0,$ then $\text{recip}\langle a, p \rangle =_A \text{recip}\langle a', q \rangle$

- The value of recip$\langle a, p\rangle$ does not depend on the actual p
- One only has to ascertain that such a term exists.

Proof Irrelevance principle: for an object $t(p) : A$, with $p : \varphi$ a sub-term of t,

$$t(p) = t(q) \text{ for all } p, q : \varphi$$

The setoid equality obeys this principle.

Intensionality versus Extensionality

The side condition equality in the conv rule is called the definitional equality

It 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 $H : (A{\rightarrow}B){\rightarrow}\text{Prop}$ 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.