Canonical Structures

Assia Mahboubi

13 March 2012

MAP INTERNATIONAL SPRING SCHOOL ON FORMALIZATION OF MATHEMATICS 2012

SOPHIA ANTIPOLIS, FRANCE / 12-16 MARCH

Inria informatics mathematics · ihp Institut Henri Poincaré · MICROSOFT RESEARCH INRIA JOINT CENTRE

Comprehension

Collections of objects satisfying a common requirement P are usually denoted by:

$$\{x \mid P(x) \text{ holds}\}$$

Examples:

- $\{n \mid n \text{ is smaller than } m\}$
- $\{l \mid l \text{ is duplicate-free}\}$
- ...

Dependent pairs

In order to model: `{n : nat | n < 5}`, we forge a type whose elements `t` are pairs:

- The first component of `t` is a natural number `n`;
- The second component of `t` is a proof that `n < 5`.

This is called a dependent pair since:
(the type of) the second component depends on the first.

Dependent pairs

In order to model: `{n : nat | n < 5}`, we forge a type whose elements `t` are pairs:

In Coq this can be defined as:

```
Inductive I5 :=
    Ordinal5 : forall m : nat, m < 5 -> I5.
```

which is exactly the same as:

```
Inductive I5 := Ordinal5 m of (m < 5).
```

Dependent pairs

This is the definition of the type:

```
Inductive I5 := Ordinal5 m of (m < 5).
```

This is an example of definition of one of its elements:

```
Variables (x : nat)(px5 : x < 5).

Definition i5x : I5 := @Ordinal5 x px5.
```

Finite ordinals

The type `ordinal` of finite ordinals is defined as:

```
Inductive ordinal (n : nat) := Ordinal m of m < n
```

and `(ordinal n)` models the natural numbers smaller that `n`.

Equality of dependent pairs

$$\{n \ : \ nat \mid n < 5\}$$

- An inhabitant t_n of this type is a pair (n, p_n)
- Comparing two inhabitants t_1 and t_2 means comparing them component-wise:
$$t_1 = t_2 \quad \text{iff} \quad (n_1 = n_2) \wedge (p_{n_1} = p_{n_2})$$
- The proof component should be irrelevant here.

But in Coq this is not true in general...

Equality of boolean values

Fortunately, we can prove the theorem for `bool`:

$$\forall\ (x\ y\ :\ bool)\ (p_1\ p_2\ :\ x = y),\ p_1 = p_2$$

And now:

- In the library `(n < 5)` has type `bool`.

Equality of boolean values

Fortunately, we can prove the theorem for `bool`:

$$\forall\ (x\ y\ :\ bool)\ (p_1\ p_2\ :\ x = y),\ p_1 = p_2$$

And now:

- In the library `(n < 5)` has type `bool`.
- Now the sigma type is: $\{n\ :\ nat \mid (n < 5) = \mathrm{true}\}$

Equality of boolean values

Fortunately, we can prove the theorem for `bool`:

$$\forall\ (x\ y\ :\ bool)\ (p_1\ p_2\ :\ x = y),\ p_1 = p_2$$

And now:

- In the library `(n < 5)` has type `bool`.
- Now the sigma type is: $\{n\ :\ nat \mid (n < 5) = \mathrm{true}\}$
- Compare (n_1, p_1) with (n_2, p_2) when $n_1 = n_2$.
  - $p_2\ :\ (n_2 < 5)\ =\ true$.
  - $p_1\ :\ (n_1 < 5)\ =\ true$.

Equality of boolean values

Fortunately, we can prove the theorem for `bool`:

$$\forall\ (x\ y\ :\ bool)\ (p_1\ p_2\ :\ x = y),\ p_1 = p_2$$

And now:

- In the library `(n < 5)` has type `bool`.
- Now the sigma type is: $\{n\ :\ nat \mid (n < 5) = \text{true}\}$
- Compare (n_1, p_1) with (n_2, p_2) when $n_1 = n_2$.
  - $p_2\ :\ (n_2 < 5)\ =\ true.$
  - $p_1\ :\ (n_2 < 5)\ =\ true.$

Equality of boolean values

Fortunately, we can prove the theorem for `bool`:

$$\forall\ (x\ y\ :\ bool)\ (p_1\ p_2\ :\ x = y),\ p_1 = p_2$$

And now:

- In the library `(n < 5)` has type `bool`.
- Now the sigma type is: $\{n\ :\ nat \mid (n < 5) = \mathrm{true}\}$
- Compare (n_1, p_1) with (n_2, p_2) when $n_1 = n_2$.
  - $p_2\ :\ (n_2 < 5)\ =\ true.$
  - $p_1\ :\ (n_2 < 5)\ =\ true.$

Using the theorem, we prove that $p_1 = p_2$ and hence $t_1 = t_2$.

Comparing inhabitants of the rich type boils down to comparing the values.

Infrastructure

Dependent pairs of type:

`{x : T | P x}` where `P : T -> bool`

are hence of special interest because of this nice property of type `bool`.

In the library, there is a special infrastructure for types that can be seen as `{x : T | P x}`, called `subType`.

More than pairs

How to deal with the abstract algebra like abstractions, as in:

"Let G be a semi-group."

meaning:
G is a set equipped with an associative binary operation.

Record types

```
Structure semiGroup := SemiGroup {
  dom : Type;
  binop : dom -> dom -> dom;
  binopA : forall x y z : dom,
        binop x (binop y z) = binop (binop x y) z
}.
```

In Coq, record types generalize the previous dependent pairs

Record types

```
Structure semiGroup := SemiGroup {
  dom :  Type;
  binop :  dom -> dom -> dom;
  binopA : forall x y z :  dom,
      binop x (binop y z) = binop (binop x y) z
}.
```

- semiGroup is the name of the type

Record types

```
Structure semiGroup := SemiGroup {
  dom :  Type;
  binop :  dom -> dom -> dom;
  binopA : forall x y z :  dom,
      binop x (binop y z) = binop (binop x y) z
}.
```

- semiGroup is the name of the type
- SemiGroup is the name of the constructor

Record types

```
Structure semiGroup := SemiGroup {
  dom :  Type;
  binop :  dom -> dom -> dom;
  binopA : forall x y z :  dom,
      binop x (binop y z) = binop (binop x y) z
}.
```

- semiGroup is the name of the type
- SemiGroup is the name of the constructor

The constructor builds new objects of type semiGroup.

Record types

```
Structure semiGroup := SemiGroup {
  dom : Type;
  binop : dom -> dom -> dom;
  binopA : forall x y z : dom,
        binop x (binop y z) = binop (binop x y) z
}.
```

```
  Definition natSemiGroup : semiGroup :=
     @SemiGroup nat addn addnA.
```

Record types

```
Structure semiGroup := SemiGroup {
  dom : Type;
  binop : dom -> dom -> dom;
  binopA : forall x y z : dom,
        binop x (binop y z) = binop (binop x y) z
}.
```

```
  Definition natSemiGroup : semiGroup :=
     SemiGroup addnA.
```

Record types

`dom`, `binop` and `binopA` are called projections.

```
Structure semiGroup := SemiGroup {
  dom : Type;
  binop : dom -> dom -> dom;
  binopA : associative binop
}.
```

Record types

`dom`, `binop` and `binopA` are called projections.

```
Structure semiGroup := SemiGroup {
  dom : Type;
  binop : dom -> dom -> dom;
  binopA : associative binop
}.
```

Hence:

- `(dom natSemiGroup)` is `nat`
- `(binop natSemiGroup)` is `addn`
- `(binopA natSemiGroup)` is `addnA`

Records as interfaces

Record types can be used as rich interfaces, in order to abstract notations and properties shared by the instances.

Let us play with a toy example.

Records as interfaces

Record types can be used as rich interfaces, in order to abstract notations and properties shared by the instances.

Let us play with a toy example.

And we quickly meet the limitations of this approach...

Unification

Unification happens when the system has to figure out that two terms are the same.

```
g : nat -> nat
c : nat
d : nat
ha : a = 0                                rewrite ha.
hbd : g b = g d
=================================
g (if (a == 0) then b else c) = g d
```

Unification

Unification happens when the system has to figure out that two terms are the same.

```
g : nat -> nat
c : nat
d : nat
ha : a = 0
hbd : g b = g d
=================================
g (if (0 == 0) then b else c) = g d
```

Unification

Unification happens when the system has to figure out that two terms are the same.

```
g : nat -> nat
c : nat
d : nat
ha : a = 0
hbd : g b = g d
=================================
g (if (0 == 0) then b else c) = g d
```

```
rewrite hbd.
```

Unification

Unification happens when the system has to figure out that two terms are the same.

```
g : nat -> nat
c : nat
d : nat
ha : a = 0
hbd : g b = g d
================
g d = g d
```

Unification

It also happens when the system has to figure out that terms with holes can be identified (finding a substitution).

```
a : nat
b : nat
==========================
(a + b).+1 - (a + b) = 1
```

```
rewrite subSnn.
```

```
Lemma subSnn : forall n : nat, n.+1 - n = 1.
```

Hints for unification

It also happens when the system has to figure out that two terms with holes can be identified.

```
a : nat
b : nat
==========================
1 = 1
```

```
rewrite subSnn.
```

```
Lemma subSnn : forall n : nat, n.+1 - n = 1.
```

Unification

Unification happens when the system has to figure out that two terms are the same.

```
c : nat
d : nat
=================
c + d = d + c
```

```
rewrite addnC.
```

```
Lemma addnC : commutative addn.

Definition commutative S T (op : S -> S -> T) :=
  forall x y, op x y = op y x.
```

Unification

Unification happens when the system has to figure out that two terms are the same.

```
c : nat
d : nat
=================
d + c = d + c
```

```
Lemma addnC : commutative addn.
```

```
Definition commutative S T (op : S -> S -> T) :=
  forall x y, op x y = op y x.
```

Unification

Back to the previous failure example:

```
x : nat
y : nat
z : nat                              apply: binopA.
=======================
x + (y + z) = x + y + z
```

```
binopA : forall s : semiGroup, associative binop
```

which expands to:

```
forall (s : semiGroup)(x y z : dom s),
binop s x (binop s x y) = binop s (binop s x y) z
```

Unification

In order to succeed, the system should find a way to identify:

```
addn      x (addn    y  z) = addn     (addn    x  y) z

binop ?   x (binop ? y  z) = binop ?  (binop ? x  y) z
```

Unification

In order to succeed, the system should find a way to identify:

```
addn      x (addn    y  z) = addn     (addn    x  y) z
binop ?   x (binop ? y  z) = binop ?  (binop ? x  y) z
```

Unification

In order to succeed, the system should find a way to identify:

```
addn    x (addn    y  z) = addn    (addn    x  y) z
binop ? x (binop ? y  z) = binop ? (binop ? x  y) z
```

Unification

In order to succeed, the system should find a way to identify:

`addn`	`x`	`(`	`addn`	`y`	`z`	`)`	`=`	`addn`	`(`	`addn`	`x`	`y`	`)`	`z`
`binop ?`	`x`	`(`	`binop ?`	`y`	`z`	`)`	`=`	`binop ?`	`(`	`binop ?`	`x`	`y`	`)`	`z`

hence it should identify:

and	`addn`	`:`	`nat`	`->`	`nat`	`->`	`nat`
	`binop ?`	`:`	`dom ?`	`->`	`dom ?`	`->`	`dom ?`

Unification

In order to succeed, the system should find a way to identify:

and

```
   addn : nat   -> nat   -> nat
binop ? : dom ? -> dom ? -> dom ?
```

But there is no way to invent such a `(? : semiGroup)`...

Inference of structures

Let us register the concrete instance we found as a `Canonical` instance.

Previously we had:

```
Definition natSemiGroup : semiGroup :=
  SemiGroup addnA.
```

And now we turn this into:

```
Canonical natSemiGroup : semiGroup :=
  SemiGroup addnA.
```

Hints for unification

The `Canonical` data-base provides extra information for problems involving a record projection:

```
addn     x (addn     y z) = addn     (addn     x y) z
binop ?  x (binop ?  y z) = binop ?  (binop ?  x y) z
```

The system should identify:

and

```
addn    : nat   -> nat   -> nat
binop ? : dom ? -> dom ? -> dom ?
```

But we have registered a canonical solution for this problem:

$$\texttt{binop ?} \equiv \texttt{addn} \quad \Rightarrow \quad \texttt{?} \equiv \texttt{natSemiGroup}$$

Hints for unification

The `Canonical` data-base provides extra information for problems involving a record projection:

```
Structure my_struct := MyStruct {
p1 : T1;
p2 : T2;
...}.
```

Hints for unification

The `Canonical` data-base provides extra information for problems involving a record projection:

```
Structure my_struct := MyStruct {
p1 : T1;
p2 : T2;
...}.

Canonical my_instance : my_struct :=
   Mystruct my_t1 my_t2 ....
```

Hints for unification

The Canonical data-base provides extra information for problems involving a record projection:

```
Structure my_struct := MyStruct {
p1 : T1;
p2 : T2;
...}.

Canonical my_instance : my_struct :=
   Mystruct my_t1 my_t2 ....
```

stores the canonical solutions to the unification problems:

$$
\begin{array}{lcl}
\mathtt{p1}\ ?\equiv \mathtt{my\_t1} & \Rightarrow & ?\equiv \mathtt{my\_struct} \\
\mathtt{p2}\ ?\equiv \mathtt{my\_t2} & \Rightarrow & ?\equiv \mathtt{my\_struct} \\
 & \ldots &
\end{array}
$$

From the library: the eqType structure

```
Structure eqType := EqType {
sort : Type;
eq_op : sort -> sort -> bool;
_ : forall x y, reflect (x = y) (eq_op x y)}.

Notation "x == y" := (eq_op x y).
```

This makes the notation `(_ == _)` available and shared by all the declared instances of eqType.

Combining structures

A `Canonical` declaration can also consist in a generic pattern for the construction of new instances from generic ones:

- The type of pairs of eqType has a canonical structure of eqType
- The type of lists of eqType has a canonical structure of eqType
- ...

Canonical instances of a structure share notations and theory.

Conclusion

- record types are used as interfaces
- unification and hence type inference can be aided by the canonical structures mechanism.
- in fact you can program this like in a prolog engine
- it is a very powerful mean of generic programming inside the proof assistant
- see tomorrow lessons on big operations and the algebraic hierarchy