Introduction to Interactive Proof of Software

J.-F. Monin

Univ. Joseph Fourier and
LIAMA-FORMES, Tsinghua Univ., Beijing

2012, Semester 1

Lecture 3

Outline

Outline

Summary of previous lectures

- We manipulate tree-like data structures called terms
- All trees have a type, which are themselves trees
- Notation: term : type
- Basic way to have new types: Inductive definitions declaring the complete set of its constructors
  example: enumerated types
- Constructors may have arguments $\rightarrow$ hence trees
- Case analysis on an enumerated type (match)
- Definitions can be written directly or interactively
- In general, things are defined within an environment made of declarations variable : type
- pluging: works for all terms having the expected type
- functions of type $\forall x_1 : t_1, \ldots \forall x_n : t_n, t_{result}$
  where t_{result} may depend on $x_1 \ldots x_n$
  example: funny : $\forall r : \texttt{rgb}, \texttt{Set\_of}\ r$

Computation on trees

The whole point of computer science is computation

On trees, it means successive transformations

$$tree_0 \longrightarrow tree_1 \longrightarrow tree_2 \longrightarrow \ldots tree_n \longrightarrow \ldots$$

- all $tree_i$ have the same type
- delimited transformations (neighboring nodes involved) called reductions
- reduction order irrelevant ****
- computation always terminates ****
- therefore, all $tree_i$ have the same value

We get **stateless** programming

Reduction of a case

Example

$$\dfrac{\dfrac{}{\texttt{Set}}\ \texttt{co} \qquad \dfrac{}{\texttt{rgb}}\ \texttt{Gf} \qquad \dfrac{}{\texttt{co}}\ \texttt{R} \qquad \dfrac{}{\texttt{co}}\ \texttt{G} \qquad \dfrac{}{\texttt{co}}\ \texttt{B}}{\texttt{co}}\ \texttt{case}$$

reduces to the "second" branch : $\dfrac{}{\texttt{co}}\ \texttt{G}$

Reduction of a case

3 reductions for `rgb`

$$\frac{\dfrac{}{\mathtt{Set}}A \qquad \dfrac{}{\mathtt{rgb}}\mathtt{Rf} \qquad \genfrac{}{}{0pt}{}{=\,t_1}{A} \qquad \genfrac{}{}{0pt}{}{=\,t_2}{A} \qquad \genfrac{}{}{0pt}{}{=\,t_3}{A}}{A}\,\mathtt{case} \quad \longrightarrow \quad \genfrac{}{}{0pt}{}{=\,t_1}{A}$$

$$\frac{\dfrac{}{\mathtt{Set}}A \qquad \dfrac{}{\mathtt{rgb}}\mathtt{Gf} \qquad \genfrac{}{}{0pt}{}{=\,t_1}{A} \qquad \genfrac{}{}{0pt}{}{=\,t_2}{A} \qquad \genfrac{}{}{0pt}{}{=\,t_3}{A}}{A}\,\mathtt{case} \quad \longrightarrow \quad \genfrac{}{}{0pt}{}{=\,t_2}{A}$$

$$\frac{\dfrac{}{\mathtt{Set}}A \qquad \dfrac{}{\mathtt{rgb}}\mathtt{Bf} \qquad \genfrac{}{}{0pt}{}{=\,t_1}{A} \qquad \genfrac{}{}{0pt}{}{=\,t_2}{A} \qquad \genfrac{}{}{0pt}{}{=\,t_3}{A}}{A}\,\mathtt{case} \quad \longrightarrow \quad \genfrac{}{}{0pt}{}{=\,t_3}{A}$$

Reduction of a case

3 reductions for `rgb`

```
match Rf with
| Rf => t1
| Gf => t2
| Bf => t3
end.
```

Reduces to t_1

```
match Gf with
| Rf => t1
| Gf => t2
| Bf => t3
end.
```

Reduces to t_2

```
match Bf with
| Rf => t1
| Gf => t2
| Bf => t3
end.
```

Reduces to t_3

Outline

Definitions

```
Definition my_function : forall (x: A), B :=
  fun x => BODY
```

where *BODY* is some code depending on

- `x`
- *previous definitions*
- *BUT NOT on future definitions*
- *NOR on* `my_function`

Equality

`my_function` a = *BODY* [`x` replaced by a]

where a is any argument of type `A`

Recursive definition

```
Definition? my_function :
  forall (x: A), B :=
  fun x => BODY
```

What does it mean if *BODY* does contain occurrences of `my_function`?

Let us add an additional argument `f`, and replace all occurrences of `my_function` by `f` in *BODY*.

```
Definition Big :
  (forall (x: A), B) → (forall (x: A), B) :=
  fun f => (fun x => BODY')
```

Expected equality

```
Big my_function
  =  (fun x => BODY') [ f replaced by my_function]
  =  my_function
```

Fixpoint

In short:

```
Big my_function  =  my_function
```

We say that `my_function` is a fixpoint of `Big`

Makes sense if

the existence and unicity of a solution of the above equation is ensured

In Coq

Recursive calls must be on a

structurally smaller argument

Important application

Induction principles are special cases of fixpoints

To be understood later, when considering proof-trees and functions over proof-trees

Outline

Functions: example

```
Definition color_of : forall (r: rgb), color :=
  fun (r: rgb) =>
  match r with
  | Rf => Red
  | Gf => Green
  | Bf => Blue
  end.
```

Application: by juxtaposition

```
color_of Bf
```

Products and functions

Consider an environment containing $x : t$ (and may be other types variables) where we define a term $U_x : u$

More generally, u may depend on x.

Consider an environment containing $x : t$ (and may be other types variables) where we define

- a type u_x
- a term $U_x : u_x$

Then $\texttt{fun}\ x \Rightarrow U_x$ is a function defined **for all** x, and returning U_x each time it applied to some argument for x.

$$\texttt{fun}\ x : t \Rightarrow U_x : \quad \forall x : t, u_x$$

Application

If $f : \forall x : t, u_x$ and $A : t$
then f can be applied to A and the type of the result is u_A

Rules (general)

$$\dfrac{\begin{array}{c}\vdots\, f\\ \forall x : t, u_x\end{array} \qquad \begin{array}{c}\vdots\, A\\ t\end{array}}{u_A}\ \texttt{apply}$$

$$\dfrac{\begin{array}{c}[x : t]\\ \vdots\, U\\ u_x\end{array}}{\forall x : t, u_x}\ \texttt{fun[x]}$$

Warning: this x makes sense only in U,
i.e. is available only from $x : t$ to u_x

When the type of the result does not depend on x

$$\dfrac{\begin{array}{c}\vdots\, f\\ \forall x : t, u\end{array} \qquad \begin{array}{c}\vdots\, A\\ t\end{array}}{u}\ \texttt{apply}$$

$$\dfrac{\begin{array}{c}[x : t]\\ \vdots\, U\\ u\end{array}}{\forall x : t, u}\ \texttt{fun[x]}$$

Warning: this x makes sense only in U,
i.e. is available only from $x : t$ to u

Other syntax: $t \rightarrow u$ instead of $\forall x : t, u$

$$\dfrac{\begin{array}{c}\vdots\, f \\ t \rightarrow u\end{array} \qquad \begin{array}{c}\vdots\, A \\ t\end{array}}{u}\ \texttt{apply}$$

$$\dfrac{\begin{array}{c}[x : t] \\ \vdots\, U \\ u\end{array}}{t \rightarrow u}\ \texttt{fun[x]}$$

Warning: this x makes sense only in U,
i.e. is available only from $x : t$ to u

Example 1

```
Definition color_of : forall (r: rgb), color :=
  fun (r: rgb) =>
  match r with
  | Rf => Red
  | Gf => Green
  | Bf => Blue
  end.

Definition color_of : rgb -> color :=
  fun (r: rgb) =>
  match r with
  | Rf => Red
  | Gf => Green
  | Bf => Blue
  end.
```

Question: where `r` is available?

Example 2

```
Definition Set_of : forall (r: rgb), Set :=
  fun (r: rgb) =>
  match r with
  | Rf => rgb
  | Gf => color
  | Bf => tuple4
  end.
```

```
Definition Set_of : rgb -> Set :=
  fun (r: rgb) =>
  match r with
  | Rf => rgb
  | Gf => color
  | Bf => tuple4
  end.
```

Question: where `r` is available?

Example 3

```
Definition Set_of : rgb -> Set :=
  fun (r: rgb) =>
  match r with
  | Rf => rgb
  | Gf => color
  | Bf => tuple4
  end.

Definition funny : forall (r: rgb), Set_of r :=
  fun (r: rgb) =>
  match r with
  | Rf => Gf
  | Gf => Yellow
  | Bf => t1
  end.
```

Remark: `Yellow : Set_of Gf`
because `Set_of Gf` reduces to `color`

Outline

Reduction of function = application to an argum^t

$$\cfrac{\cfrac{\begin{matrix}[x : t] \\ \vdots\, U \\ u_x\end{matrix}}{\forall x : t, u_x}\,\texttt{fun[x]} \qquad \begin{matrix}\vdots\, A \\ t\end{matrix}}{u_A}\,\texttt{apply}
\qquad\qquad
\begin{matrix}\vdots\, A \\ t \\ \vdots\, U \\ u_A\end{matrix}$$

`(fun x => U) A` $\qquad\qquad$ `U [x := A]`

Substitution: `U [x := A]` is `U` where
all free occurrences of `x` are replaced by `A`.

Example

`Set_of Gf` reduces to

```
(fun (r: rgb) =>
  match r with
  | Rf => rgb
  | Gf => color
  | Bf => tuple4
  end)  Gf
```

reduces to

```
  match Gf with
  | Rf => rgb
  | Gf => color
  | Bf => tuple4
  end
```

reduces to `color`

Introduction, elimination, reduction

General statement from Proof Theory

In each type we have corresponding introduction and elimination rules, as well as reductions

For inductive types

- introduction = constructor
- elimination = case

Introduction, elimination, reduction work together

- Observation: reducing a tree yields a constructor at its root
- The latter can be the key argument of a case
- Therefore, case analysis on constructors is exhaustive

Outline

Functions of several arguments

In $\forall x, u_x$, u_x can itself be a product type $\forall y, v_{xy}$
We get $\forall x, \forall y, v_{xy}$ which reads $\forall x, (\forall y, v_{xy})$

Typing:

- $x : t$
- $U_x : u_x$
- $y : r_x$ (the type of y may depend on x!)

Alltogether : $\forall x : t, \forall y : r_x, v_{xy}$

In particular, $\forall x : t, r_x \to v_x$ reads $\forall x : t, (r_x \to v_x)$
and $t \to r \to v$ reads $t \to (r \to v)$

Consistently, $f\ A\ B$ reads $(f\ A)\ B$,

given $f : t \to (r \to v)$, $A : t$ and $B : r$
or $f : \forall x : t, \forall y : r_x, v_{xy}$, $A : t$ and $B : r_A$

Example: identity function (specific)

```
Definition id_rgb : forall (r: rgb), rgb :=
  fun (r: rgb) =>
  match r with
  | Rf => Rf
  | Gf => Gf
  | Bf => Bf
  end.
```

Simpler

```
Definition id_rgb : forall (x: rgb), rgb :=
  fun (x: rgb) => x.
```

Similarly

```
Definition id_color : forall (x: color), color :=
  fun (x: color) => x.
```

Example: identity function (general)


```
Definition id_rgb : forall (x: rgb), rgb :=
  fun (x: rgb) => x.
```

```
Definition id_rgb : rgb -> rgb :=
  fun (x: rgb) => x.
```

Generalization

```
Definition id : forall (X: Set),forall (x: X), X :=
  fun (X: Set) (x: X) => x.
```

```
Definition id : forall (X: Set), X -> X :=
  fun (X: Set) (x: X) => x.
```

```
Definition id_rgb : forall (x: rgb), rgb :=
  id rgb.
```

Application of a function to several arguments

```
Definition id : forall (X: Set), X -> X :=
  fun (X: Set) (x: X) => x.
```

The term `id rgb Gf` reads `(id rgb) Gf`

And similarly for functions expecting 3, 4... arguments

Constructors as functions

```
Mk4rgb : forall x1, x2, x3, x4: rgb, tuple4
Mk4rgb : rgb -> rgb -> rgb -> rgb -> tuple4
```

```
Mk4rgb Gf Rf Gf Bf
```

Partial application of a function

We have already seen: `id rgb`

What is meaning and the type of `Mk4rgb Gf Rf` ?

Functions as first class objects

We have seen that the result of a function can be a function

Similarly, a function can be passed as an argument of a function

Example: `id (rgb → color) color_of`

Exercises:

- Reduce the previous expression
- Reduce: `id (rgb → color) color_of Bf`

Conclusion on functions

Functions are one of the prominent feature of Coq, where they live in a very general setting.

In particular we will see that proofs are always trees and are even functions most of the time

Hence the importance of

- defining functions
- using functions (application)
- typing functions

Next important notions

- pattern matching
- application to logic
- recursive functions (fixpoints) and induction

Outline

Pattern matching

- The destruct tactic and the match construct in the case where constructors have arguments
- More general pattern matching
- See `lecture03.v` and friends

- Much better than Lisp or C style
- Important special case: empty inductive type

Example: lists

Here we consider list of Booleans for simplicity

```
Inductive list : Set :=
  | Nil : list
  | Cons : bool -> list -> list.
```

Scheme of use for pattern matching:

```
match l with
| Nil =>  expression_1
| Cons h t =>  expression_2 of h and t
end.
```

Why pattern matching is nice

Definition of the length of a list using pattern matching

```
Fixpoint length (l: list) : nat :=
  match l with
  | Nil => O
  | Cons h t => S (length t)
  end.
```

Compare with (in Lisp or C-like style)

```
...if beq_list l Nil then O else S (length (tail l))
```

Here, `tail` makes sense only if its argument is a non-empty list, but it is non trivial that the `else` branch of `beq_list l Nil` ensures that (the correctness of our definition of `beq_list` is questionnable).
In contrast, pattern-matching provides a comfortable environment for *expression_2*, where `h` and `t` are available with the right type for free.

Empty inductive type

An inductive may have any number of constructors, including 0.

```
Inductive empty : Set := .
```

Pattern matching: no case (0 branch) to consider:

```
Variable e: empty.
match e return nat with end.
```

Note the **return** clause in the **match** construct: it aims at providing the type of expressions on the different branches, when it cannot be guessed from the context.

Dependent inductive types

Pattern matching is still more powerful in the case of dependent inductive types

Dependent type

When a type depends on values or types provided by the current environment
Example: `funny` in previous lectures.

Inductive dependent type

See more advanced lectures

Very important special case

Equality

Dependent inductive types

Example without special meaning

```
Inductive dontcare : bool -> Set :=
  | D0 : dontcare true
  | D1 : forall (b:bool) (n: nat),
         even n -> dontcare b -> dontcare (negb b).
```

Scheme of use for pattern matching,
assuming `d: dontcare b`

```
match d with
| D0 =>  expression_1
| D1 b' n e d' => expression_2 of b', n, e and d'
end
```

Outline

Special case: equality

Theory

The notation `x = y` is a shorthand for `eq x y`, where `eq` is inductively defined.
The precise definition involves some subtelties, to be introduced later.

For practice it is much simpler

We just need:

- For all type A, and $x, y : A$,
  $x = y$ is something that we can try to prove
- Canonical proofs of equality are by reflexivity
- Destructing (i.e., using) equalities: rewrite

Equality in practice

Proving an equality

Canonical proofs of equality are by reflexivity,
a shorthand for apply `refl_equal`

$$\texttt{refl\_equal} : \forall A, \forall x : A,\ \ x = x$$

Using an equality

If

- the environment contains $e : X = Y$
- the current goal concludes to $P\ X$

Then rewrite e yields $P\ Y$

Variants:

- rewrite -> e (same effect)
- rewrite <- e (replaces $P\ Y$ by $P\ X$)