Some comments about notations and “beautification” in Coq

(June 2017)

Hugo Herbelin

Two kinds of notations

Notations modifying the parser and printer:
- e.g. `Notation "[ x ]" := (cons x nil) (at level 0, x at level 200).`
- requires parsing/printing rules (level, associativity, internal levels, printing boxes)
- are interpreted in “interpretation scopes”

Abbreviations: qualified names hiding expressions
- e.g. `Notation single x := (cons x nil).`
- they obey the general parsing rules of applications
- internally called *syntactic definition*

The processing phases from parsing to typing

(highlighting handling of notations)

$$\texttt{string/channel} \xrightarrow[\texttt{lexer.ml4/g\_}*\texttt{.ml4}]{lexing/parsing} \texttt{constr\_expr} \xrightarrow[\texttt{constrintern.ml}]{\text{``}internalization\text{''}} \texttt{glob\_constr} \xrightarrow[\texttt{pretyping.ml}]{\text{``}pretyping\text{''}} \texttt{constr}$$

lexing/parsing

- based on camlp5 (roughly LL(n) parser, development version also with backtracting)
- parsing of notations

internalization

- insertion of implicit arguments
- globalization of names
- checking binders
- interpretation of notations and abbreviations

pretyping

- type-checking and de-Bruijn-ization of binders (`pretyping/pretyping.ml`)
- resolution of implicit arguments using type classes, unification, tactics
- pattern-matching compilation (`pretyping/cases.ml`)
- insertion of coercions (`pretyping/coercion.ml`)

Relevant files for interpreting the notation commands

`vernac/metasyntax.ml`

interpret the commands `Notation`, `Delimiters`, ...

`parsing/egramcoq.ml`

declare the grammar rules

`interp/notation.ml`

the tables storing notations, scopes, printing rules, etc.

`interp/syntax_def.ml`

the tables storing abbreviations (i.e. internally syntactic definitions)

`intf/notation_term.ml`

contains `notation_constr` which is the copy of `constr` used to represent interpretation of notations (distinct from `constr` or `glob_constr` in that it contains a field for recursive patterns in notations, a field for holes, no field for (existing) existential variables, etc...)

The printing phases

(highlighting handling of notations)

$$\texttt{constr} \xrightarrow[\texttt{detyping.ml}]{\text{``}detyping''} \texttt{glob\_constr} \xrightarrow[\texttt{constrextern.ml}]{\text{``}externalization''} \texttt{constr\_expr} \xrightarrow[\texttt{pp}*\texttt{.ml}]{formatting} \texttt{std\_ppcmds} \xrightarrow{displaying} \texttt{string}\ \text{or UI}$$

detyping

- turning De Bruijn's indices into names
- partial decompilation of compiled pattern-matching

externalization

- removing implicit arguments, or turning them into explicit implicit arguments
- optimal shortening of global names
- removal of coercions
- recognizing where notations and abbreviations can be used

displaying/printing

- used OCaml's formatting machinery

Note: This is not exactly symmetrical to the typing phases (for instance, coercions are easier to remove in the externalization phase)

Relevant files for handling notations occurring in terms

`interp/notation_ops.ml`

the algorithms to interpret or recognize the pattern of a notation

- function `notation_constr_of_constr`: interpret the r.-h. s. of a notation
- function `match_notation_constr`: recognizes that an expression matches the r.-h. s. of a notation

`interp/constrintern.ml`

- entry point to interprete a notation: `intern_notation`
- function `instantiate_notation_constr`: interprets a notation applied to some instance

`interp/constrextern.ml`

- entry point to use a notation for printing: `extern_notation`

Notations: typical directions for improvements

- support for user-defined `constr` entries (EJGA, Beta)

```
Notation "'with' 'attributes' x .. y"
  := (MyConstructor (cons x .. (cons y nil) ..)
  (x mysubconstrentry, y mysubconstrentry).
Notation "'in' x" := (OfString x) (in mysubconstrentry, x string).
Notation "'of' x" := (OfInt x) (in mysubconstrentry, x int).
```

- scope-based selection of a notation at printing time (Beta, Ralf)
- support for arbitrary unary binders (rather than n-ary)
- ...

The “Beautifier”

Pre-8.0 Coq had a different syntax:

```
Theorem inst : (x:A)(all ? [x](P x))->(P x).
Proof.
Unfold all; Auto.
Qed.
```

Coq 8.0 released with an automatic translator giving here:

```
Theorem inst : forall x:A, all (fun x => P x) -> P x.
Proof.
  unfold all; auto.
Qed.
```

It was working without human intervention on the whole Coq contributions.
The “Beautifier” is a (new) name for the infrastructure which provided this translator.

The “Beautifier”

$$\begin{tikzcd}
\texttt{string/channel} \arrow[r, "lexing/parsing", "\texttt{lexer.ml4/g\_*.ml4}"'] & \texttt{vernac\_expr/tactic\_expr/constr\_expr} \arrow[d, "\text{(\textit{possibly ``beautifying'' transformations inserted here})}"] \\
\texttt{string/channel} & \texttt{vernac\_expr/tactic\_expr/constr\_expr} \arrow[l, "pretty-printing"', "\texttt{ppvernac.ml/g\_*.ml4}"]
\end{tikzcd}$$

Note: The transformations could be called directly at the level of `vernac_expr/tactic_expr/constr_expr` by UI.

Examples of possible transformations

- adding bullets in proof scripts (Théo, using an intermediate structure for manipulating proof scripts)
- adding (most) names needed in proof scripts to ensure that user occurrences of variables are user-bound

  (no miracles though, would need e.g. an `as` clause for `intuition`, etc.)
- changes around `Next Obligation`
- reindendation of scripts
- global re-printing of files using some notations (e.g. `S n` into `n.+1`, or `{x & P}` into `sigma x, P` so that it supports n-ary binders)
- translation of deprecated tactics/commands into supported ones (e.g. ‘Implicit Arguments‘ into ‘Arguments‘)
- ...