A short overview of the code handling notations in Coq

(June 2016 - Coq 8.6)

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)

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

lexing/parsing

- based on camlp4/camlp5 (roughly LL(n) parser)
- 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

`toplevel/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)

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

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`