Introducing the new Coqdoc

François Ripault
francois.ripault@epita.fr

18 février 2013

Introduction

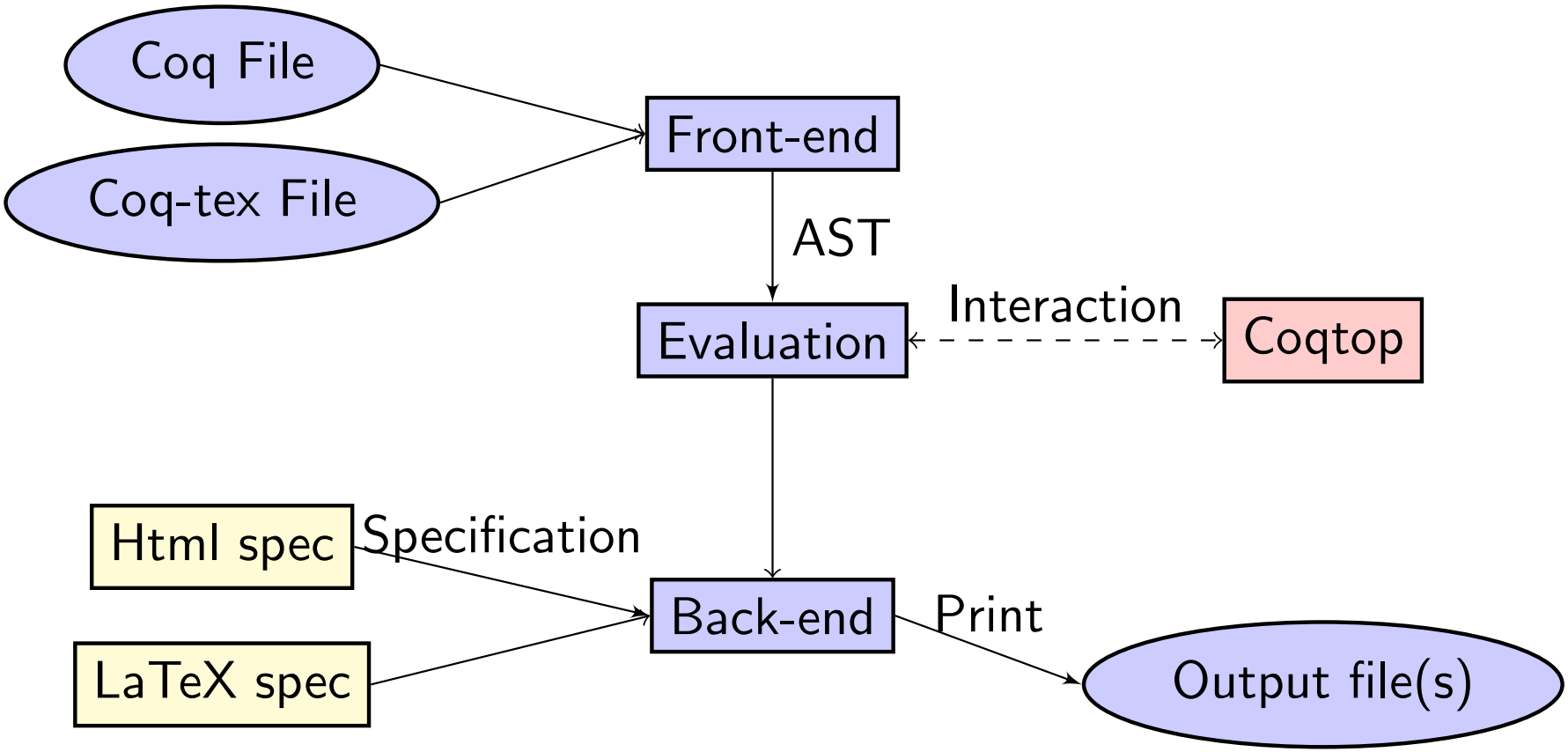
What is Coqdoc ?

- Documentation tool for Coq
- Many possible use cases

Why a new Coqdoc ?

- Coqdoc is hard to maintain
- Lack of extensibility
- Better integration of the tool

Coqdoc-ng's architecture



Coqdoc's input files

```
(** * Example_doc
  This is an example file

----

  Here is a list :
  - With a first item
  - And a second
*)

(* Now we write some code ! *)
Definition id (n : nat) := n.

(** This returns the _value 42_ *)
Eval compute in id 42.
```

Front-end

Role :

- Translation into an abstract representation
- Multiple parsers and lexers
- Selected parser depends on the input file/option

Two processing phases :

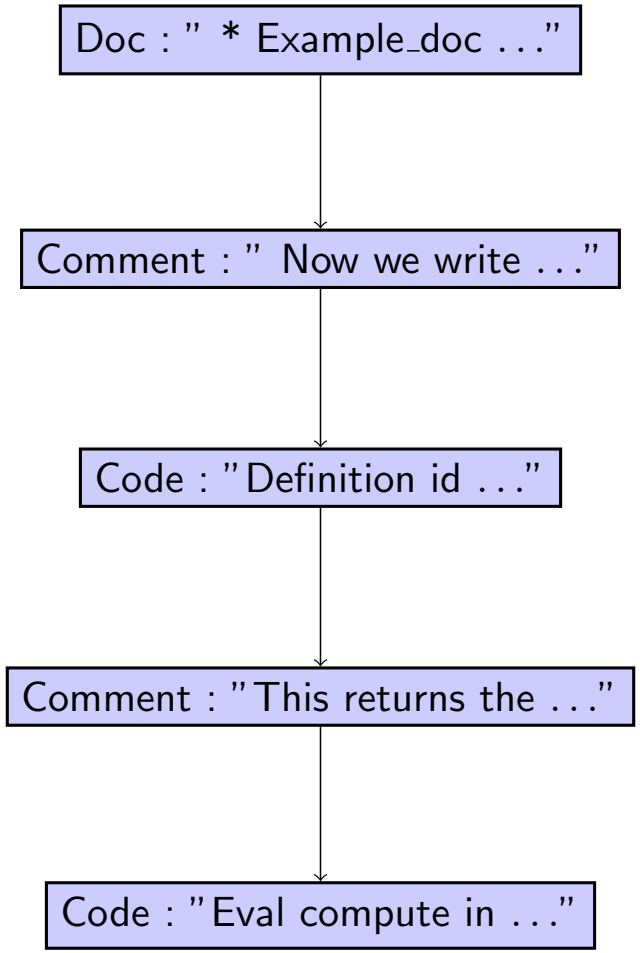
1. Separation between comments, documentation and code
2. Documentation processing

Front-end

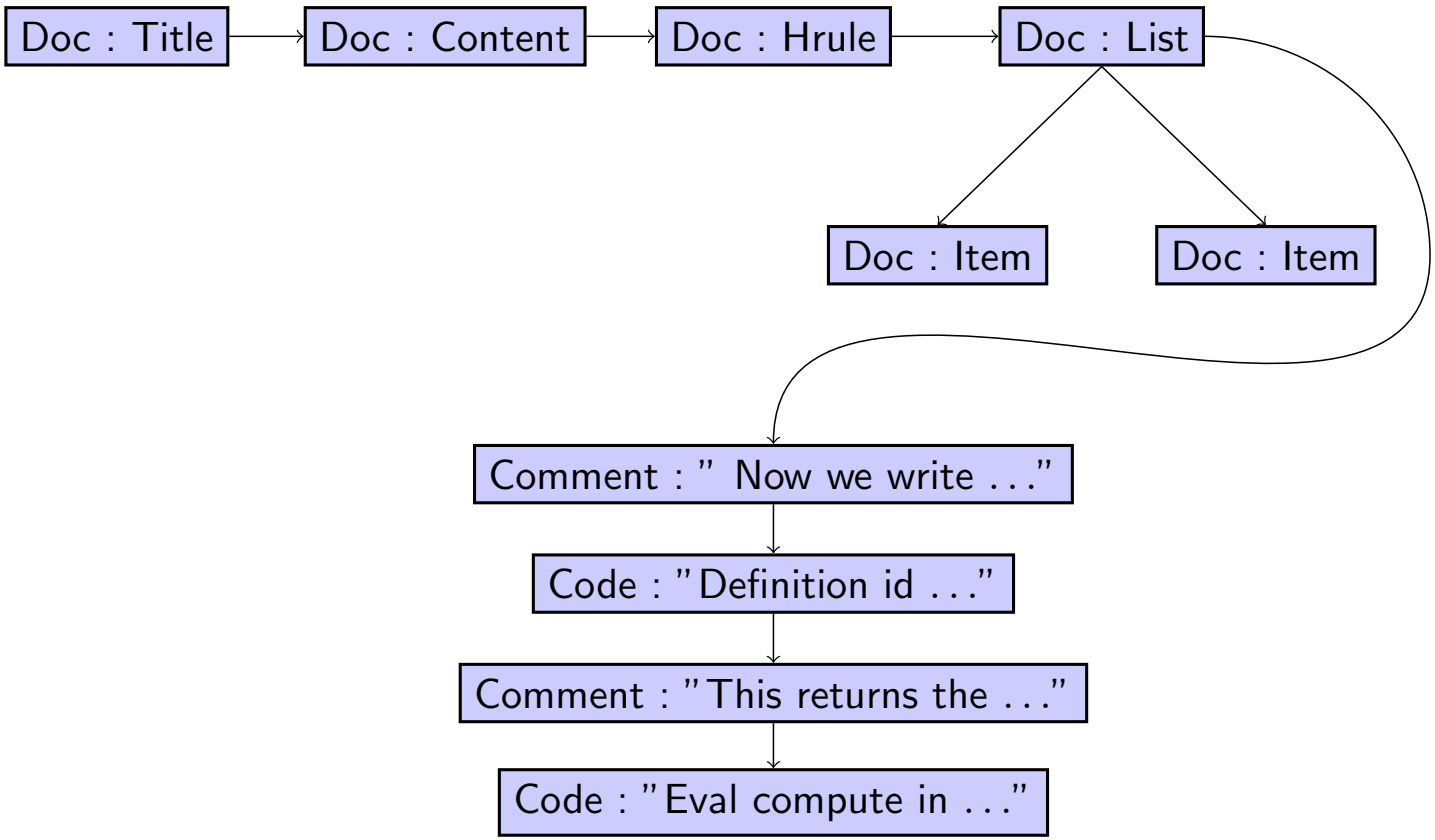
Abstract representation :

- Comments : `string`
- Code : `string` (for now ...)
- Documentation :
  - Simple elements (content, title, horizontal rule)
  - Recursive elements (lists, emphasis)
  - Elements subject to evaluation (query, printing rule)

Our document



After parsing the documentation



Evaluation

Translation for each type :

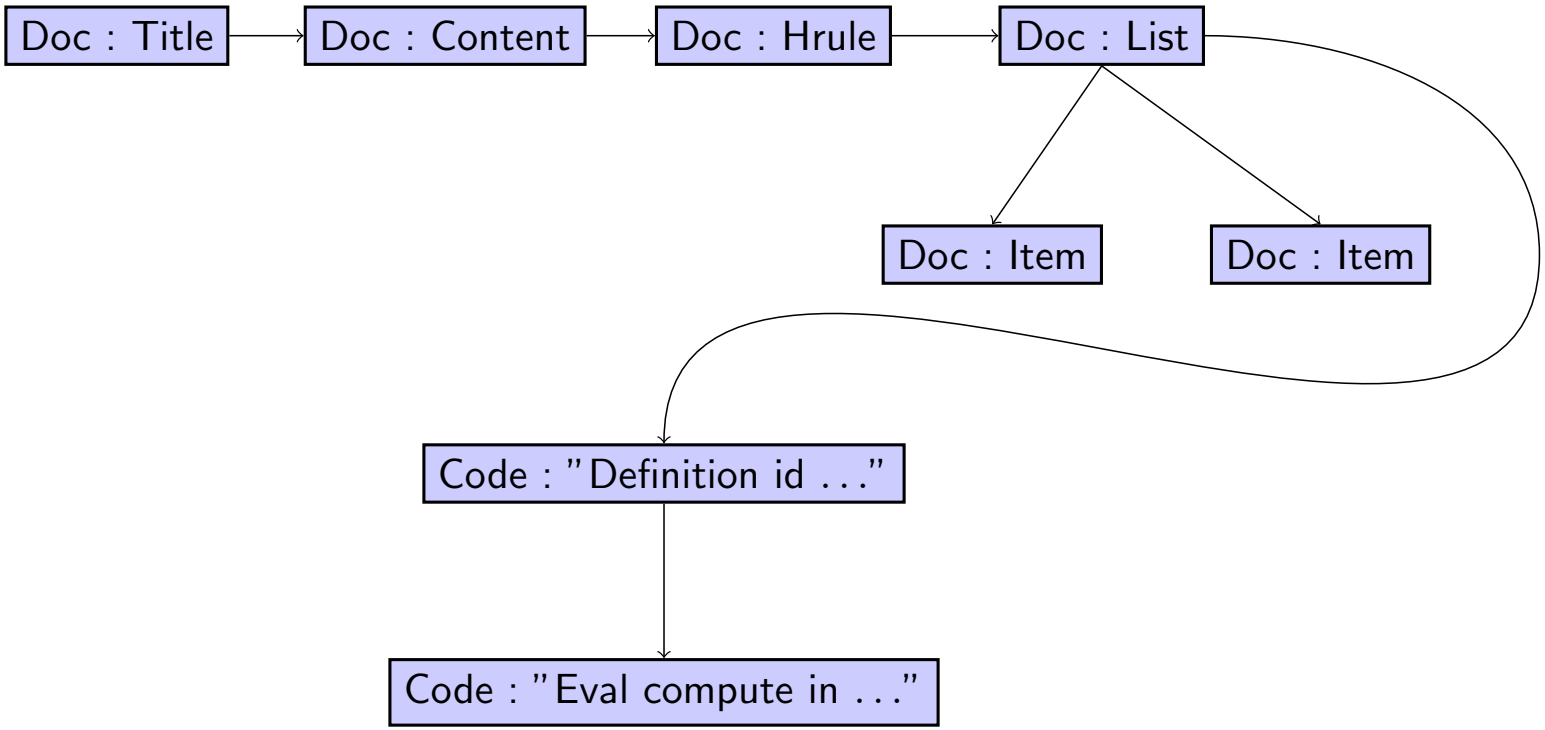
- Comments :
  - Treat show commands : set a global state

```
(* begin hide *)
  some code ...
(* end hide*)
```

  - Remove every comment
- Documentation : Translate elements subject to evaluation
  - Query : apply the function

```
(** @query{some,arguments} *)
```

  - Printing rule : set a new printing rule in the global state
  - Apply the printing rules on the rest
- Code : interaction with Coqtop


Doc : Title
Doc : Content
Doc : Hrule
Doc : List
Doc : Item
Doc : Item
Code : "Definition id ..."
Code : "Eval compute in ..."

Interaction with Coqtop

Why interact with Coqtop ?

- Remove code processing in Coqdoc
- Xml protocol (already used by CoqIDE)
- Easy to add new mechanisms

Objectives for code processing :

- Indentation
- Syntactic coloration
- Identifier processing
- Notations processing

New commands : `locate` and `prettyprint`

- `locate` : locates an identifier
  - From a name, returns its "absolute" name
  - Generate hyperlinks between different modules being documented
  - Easy to implement

- `prettyprint` : indents and annotates code
  - Coqtop indents and tags each element
  - Coqdoc translates the tags into final elements (syntactic coloration, processing of identifiers, ...)
  - Hard to do : no good level of abstraction (lack of a concrete syntax tree)

An imperfect solution for Prettyprint

Method :

- Use Coq's `Printing` module
- Annotate the output with tags
- Semantic kept and indentation obtained

Implementation :

- Make tags based on AST types (VernacExpr and ConstrExpr)
- Each node translation is surrounded with a tag function
- Generates Xml content carrying the tags

Code modification example

Modification of the printing module (in red) :

```
(* some code *)
| CApp (_,(None,a),l) ->
  tag C_CApp (pr_app (pr mt) a l), lapp
(* some code *)
```

The tag function surrounds the output with XML tags
Example output :

```
<checkmayeval>Eval compute in
  <cnotation>
    <unpmetavar> <prim>5</prim></unpmetavar>
    <unpterminal> +</unpterminal>
    <unpmetavar><prim>5</prim></unpmetavar>
  </cnotation>
</checkmayeval>.
```

Processing rules for the code

Main tasks :

- Annotate code for syntactic coloration
- Apply printing rules
- Process identifiers

Chain of responsibility design pattern :

- Apply a set of commands to a data
- Each command is a function taking the next command to apply as an argument

Advantages :

- Easy to add rules
- Extensible
- Possibility to chain commands

Output : better type for representing code

Code type

Three main code types :

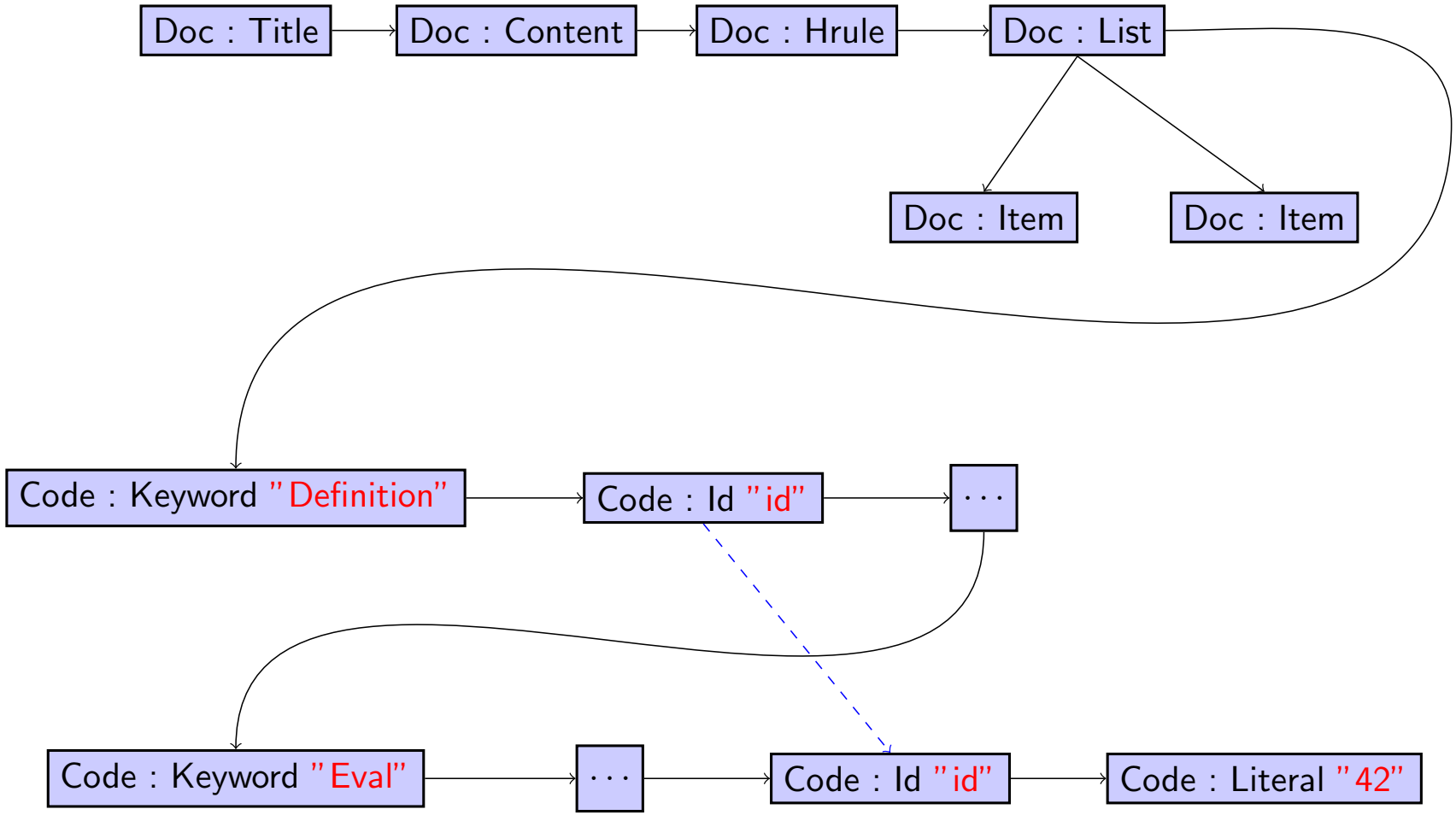
- Formatting : keywords, literals, tactics, ...
- Identifiers : contain hyperlinks
- No format

Identifiers

Identifier processing chain :

- Use `locate` command on each identifier
- Ignore undeclared identifiers
- Special cases for standard library or external library
- Generate a link type used in the output module.

Code and hyperlinks Result



Back-end

Back-end structure

- Central module in charge of printing
- Specification modules for each output type
  - Printing functions for each type (documentation and code)
  - Hyperlink generation function
  - Index generation functions

Extending the notations

A richer notations system :

- Generate output specific commands for the code the code : `sum(a,b)` becomes

```
\sum^{a}_{b}
```

New file format : Vdoc

A new file format : `.vdoc`

- Portable file : back-end independent
- Saved after evaluation
- Used to generate Coqdoc's documents
- Better control over generated files

Coq-tex

How coq-tex works :

- Code in 3 environments :
  - `coq_example` : print input and result of evaluation
  - `coq_example*` : only print input
  - `coq_eval` : only print the result of evaluation
- Everything else is left unprocessed

Parts to add :

- Add a new front-end : each environment is translated into a query type
- Process queries
- Modify the LaTeX back-end
- Add Coq-tex options to Coqdoc

Demo

Demo

Remaining work

What's left to do :

- Bugfixes
- Retro-compatibility (mostly options)
- Clean errors

Two scopes of improvement :

- XML protocol
- Coqdoc

Conclusion

State of Coqdoc :

- Mostly working prototype
- Little work left to do before release
- Space for enrichment

Questions ?