

Adapting Coq-Lsp for Lambdapi



A brief history of Lambdapi Lsp

- ✦ *The code is in the same repository of Lambdapi*
- ✦ *The Server is in src/lsp and is written in Ocaml, installs with opam*
- ✦ *The clients are in the editors folder.*
- ✦ *The focus is currently on Vscode extension written in Typescript (extension for Emacs and VI exist also).*
- ✦ *Can be installed from Vscode marketplace.*

A brief history of Lambdapi Lsp

- ✦ *Lambdapi v.2.5.1 released on July, 22nd 2024*
 - ✦ *Bug fixes*
 - ✦ *Enhance messages in the terminal and navigation in document*
- ✦ *Vscode extension : V0.2.2 released on may 16th 2024*
 - ✦ *Bug fixes*
 - ✦ *Use latest Vscode librairies*
 - ✦ *Reopen the goals panel or bring it back to front when navigation proof*

Coq-Lsp

What is it about?

- ✦ *Coq-Lsp is an Lsp server for Coq with interesting features to interactively develop proofs in Coq (incremental compilation, Document outline, ...)*
- ✦ *LambdaPi and Coq are very similar ...*
- ✦ *Why not reuse the Coq-Lsp code so to benefit from the features it implements*
- ✦ *Moreover, on the long term, it is more efficient to rely on Coq-Lsp to benefit from the existing support, maintenance, ... the Coq-lsp community offers.*

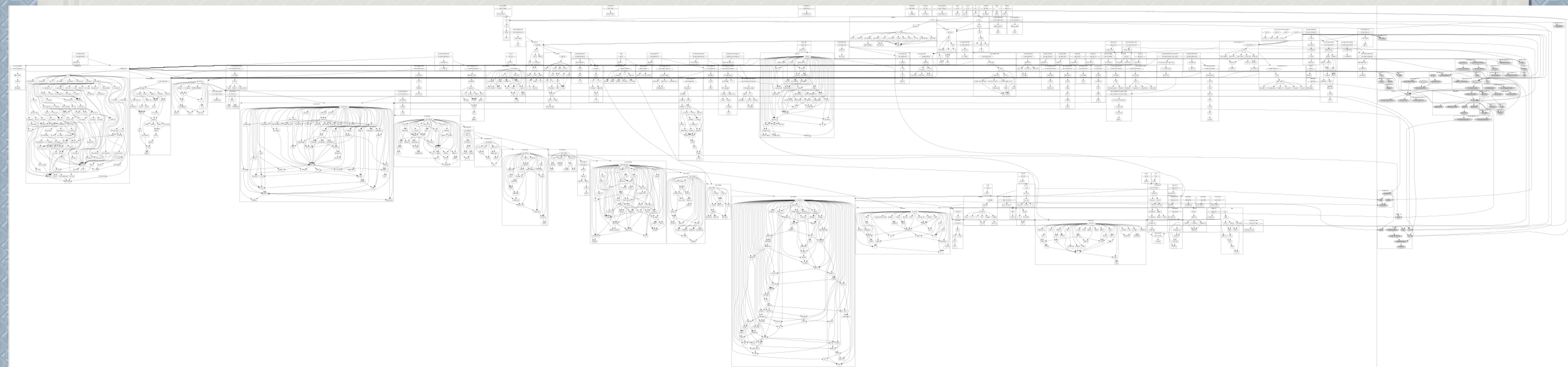
Work plan

- ✦ *Identify the modules of Coq-Lsp, their inter-communication API and the global API*
- ✦ *Refactor the code to make it adaptable with Lambdapi :*
 - ✦ *Separate commun code and specific code*
 - ✦ *Ideally, move the specific code to the modules in the bottom layer of architecture (the ones that directly interact with the prover (Coq or Lambdapi))*
- ✦ *Write the modules specific to Lambdapi*
- ✦ *Write the Glue code : the code that selects the right specific modules*

How to identify functional modules and API in OCaml

- ✦ *Specifically, understanding the Non functional aspects of the code.*
- ✦ *Some text-based tools exist to analyse OCaml code and extract information*
 - ✦ *dune-deps, not-ocamlfind, depgraph, module-graph, odoc-depgraph, ...*

Odep to understand dependencies



- ✿ *Quit rudimentary*
- ✿ *Not all useful information (at least the one that interest me) is extracted.*
- ✿ *Information is not well presented : no customization, overloaded, ...*

Code Analysis and visualization

The screenshot displays a code analysis tool interface. On the left, a tree view shows a hierarchy of entities under 'FAMIX.NamedEntity'. The 'FAMIX.Type' entity is selected and highlighted in blue. Below it, a table lists various attributes and their types.

name	type
annotationInstances	NamedEntity
argumentsInParameterizedTypes	ParameterizedType * /
attributes	Attribute * /
behavioursWithDeclaredType	BehaviouralEntity * /
belongsTo	NamedEntity
clientTypes	ContainerEntity /
comments	SourcedEntity
container	ContainerEntity
declaredSourceLanguage	SourcedEntity
definedAnnotationTypes	ContainerEntity
functions	ContainerEntity
incomingReferences	ContainerEntity
instances	Instance * /
methods	Method * /
outgoingReferences	ContainerEntity

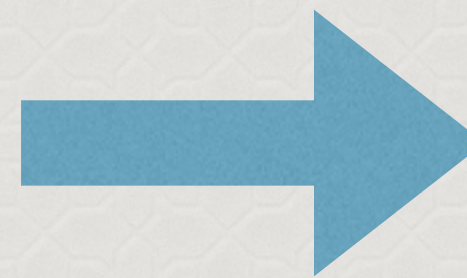
Below the table, a 'Map' visualization shows a complex network of nodes and edges. A central node is highlighted in blue, and several edges are highlighted in red, indicating relationships between different parts of the code analysis.

How to identify functional modules and API in OCaml

- ✦ *By hand*
 - ✦ *Read code*
 - ✦ *Use .mli files*
 - ✦ *Extract APIs and determine dependencies*
 - ✦ *Cumbersome for large code*
- ✦ *TIP : change the Dune files and let the compiler highlight the dependencies*

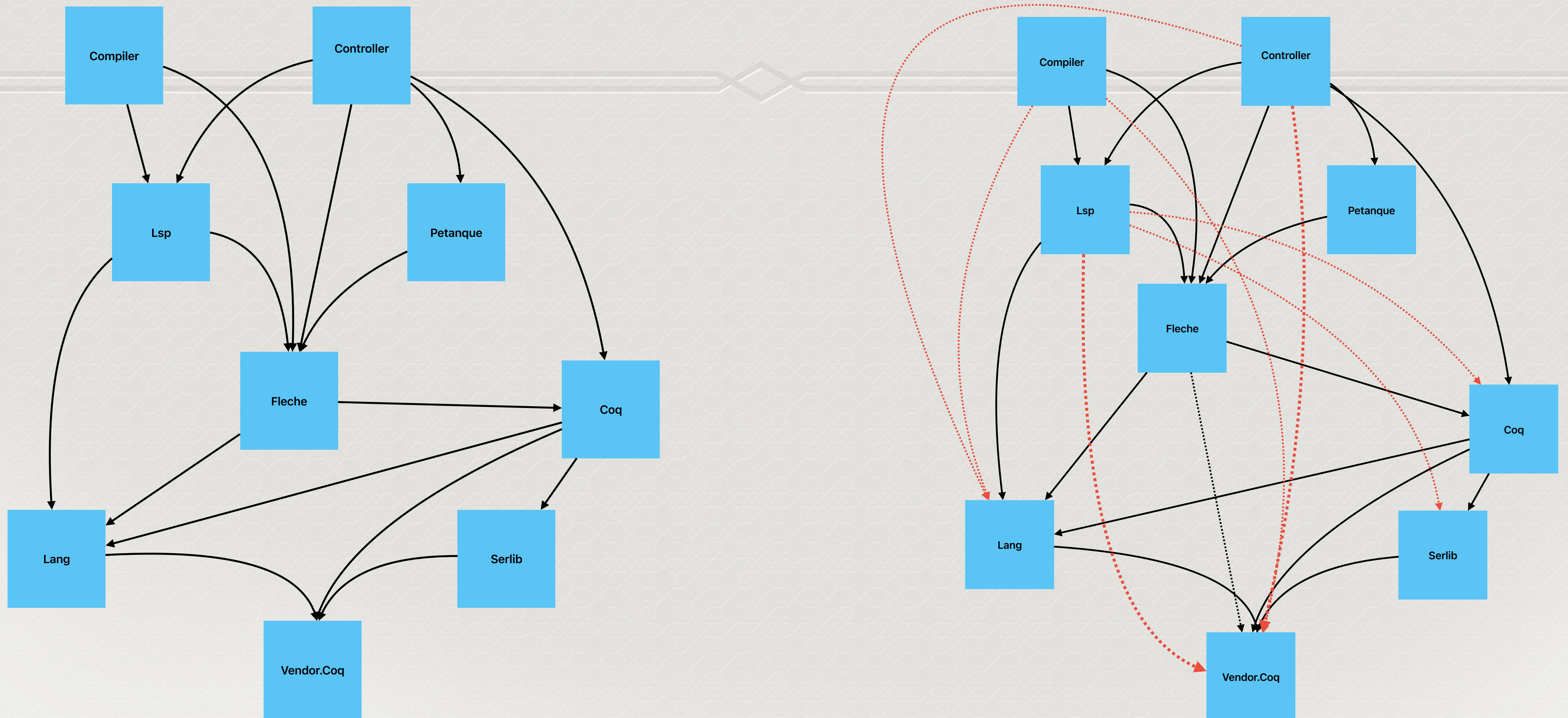
Detecting dependencies

```
doc.ml  dune coq-lsp · compiler M  dune coq-lsp · coq M ×
coq-lsp > coq > dune
1  (library
2  (name coq)
3  (public_name coq-lsp.coq)
4  (preprocess
5  (pps ppx_compare ppx_hash))
6  (libraries
7  (select
8  limits_mp_impl.ml
9  from
10 (memprof-limits -> limits_mp_impl.real.ml)
11 (!memprof-limits -> limits_mp_impl.fake.ml))
12 lang
13 ; -coq-core.vernac
14 coq-lsp.serlib
```



```
CONTRIBUTING.md  ast.ml 9+  vernacexpr.mli  dune ml mli ...
coq-lsp > fleche > doc.ml <T> t > toc
270 (** A Flèche document is basically a [node list], which is a cr
271     meta-data map [Range.t -> data], where for now [data] is th
272     [Node.t]. *)
273 type t =
274   { uri : Lang.LUri.File.t (** [uri] of the document *)
275     ; version : int (** [version] of the document *)
276     ; contents : Contents.t (** [contents] of the document *)
277     ; nodes : Node.t list (** List of document nodes *)
278     ; completed : Completion.t
279     (** Status of the document, usually either completed, s
280     waiting for some IO / external event *)
281     ; toc : Node.t CString.Map.t (** table of contents *)
282     ; env : Env.t (** External document enviroment *)
283     ; root : Coq.State.t
284     (** [root] contains the first state document state, obt
285     a workspace to Coq's initial state *)
state.ml 9+ ×
coq-lsp > coq > state.ml <T> t
1  type t = Vernacstate.t
2
3  (* EJGA: This requires patches to Coq, they are in the lsp_debug
4
5  let any_out oc (a : Summary.Frozen.any) = (* let (Summary.Fro
6  _value)) = a in *) (* let name = Summary.Dyn.repr tag in *) (
7  Lsp.IO.log_error "marshall" name; *) Marshal.to_channel oc a
```

The package diagram of Coq-Lsp



Refactoring the code

```
type fleche_document_type = {  
  ... (* independent from prover *)  
}
```

```
module type ProverDocument = sig  
  type fleche_document_type  
  type prover_document_type  
  val specific_function :  
end
```

```
module FlecheDocumentFunctor =  
functor(P:ProverDocument) -> struct  
  type t = P.t  
  type prover_document_type = P.prover_document_type  
  let specific_function = P.specific_function  
  let comun_function = (* Commun code here *)  
end
```

```
type lambda_pi_specific_type = {  
  ...  
}
```

```
module LambdaPiDocument = struct  
  type t = comun_t  
  type source = LambdaPi.LambdaPiDocument.t  
  let specific_lambda_pi_function = ...  
end
```

```
module FlecheDocumentForLambdaPi = FlecheDocumentFunctor(LambdaPiDocument)  
FlecheDocumentForLambdaPi.specific_function ...  
FlecheDocumentForLambdaPi.commun_function ...
```

Lessons learnt

- ✦ *Open projects are great*
- ✦ *A gap may exist between what the Readme says and what the code looks like.*
- ✦ *Investing in open code can pay at the mid and long terms. But,*
 - ✦ *Can be non negligible.*
 - ✦ *Many parameters determine if it worths it :*
 - ✦ *Quality of code*
 - ✦ *Documentation*
 - ✦ *Community engagement in the project and openness*

Future of development

- ✦ *Coq-Lsp developers are willing to evolve it to work with other Provers*
- ✦ *What can be done on our side :*
 - ✦ *Leave it to them and focus on LambdapiPi specific code*
 - ✦ *Contribute to the refactoring : Documentation (models), specific and non specific code.*
 - ✦ *Fork the project.*



Thank you
kindly

Questions?