Meta-programming in OCaml
Ppxlib: How We Got Here and
Where We Are Now

Paul-Elliot Anglès d'Auriac, Carl Eastlund, Sonja Heinze

June 1, 2023

Abstract

About nine years ago, the OCaml community let go of the "maintenance hell" of source code meta-programming (Camlp4/5). Since then, meta-programming is mostly done on the parsetree and has incrementally evolved into a solid ecosystem, enriching the OCaml developer experience. Two highlights of that evolvement: introducing parsetree migrations to gain individual cross-compiler compatibility (OMP); and orchestrating those migrations to create an up-to-date ecosystem with clear composition semantics and good performance (ppxlib).

In this talk, we outline the mentioned evolvement and explain the remaining challenges, i.e. creating a good higienic meta-programming culture and offering the best in terms of stability.

1 Introduction

Meta-programming enhances the programming language experience by opening the door to features such as boilerplate generation, conditional compilation, domain-specific syntax extensions, or generated file inclusion. It can be done on any representation of the program.

As a statically typed language, OCaml successively constructs two in-memory representations of the raw representation, i.e. the source-code.

$$\textit{source} \xrightarrow{\text{parsing}} \textit{parsetree} \xrightarrow{\text{typing}} \textit{typedtree}$$

Nowadays, the clear standard for OCaml meta-programming is parsetree focused and is done via a compiler-external package called *ppxlib*. As of today, $\sim$ 2000 OCaml packages out of $\sim$ 4200 OCaml packages in total (on *opam*) depend transitively on *ppxlib*.

2 Historic overview

On the way to that standard, we've had a long history of OCaml meta-programming facilities.

2.1 Source-code based

The initial approach was source-code focused. It worked by extending the parser to one's needs[2]:

$$\textit{source} \xrightarrow[\text{parsing}]{\text{extended}} \textit{parsetree} \xrightarrow{\text{typing}} \textit{typedtree}$$

Among many other problems, maintaining that parallel parser with extensible grammar was a huge maintenance burden.

2.2 Parsetree based

That's why in 2014 the approach became parsetree (*AST* from now on) focused with *ocaml*.4.02.00. The AST was augmented by two meta-programming dedicated nodes[3][1]. Meta-programming tools, called PPXs, extend those nodes to other nodes:

$$\textit{source} \xrightarrow{\text{parsing}} \textit{AST} \xrightarrow{\text{PPXs}}$$
$$\to \textit{expanded AST} \xrightarrow{\text{typing}} \textit{typedtree}$$

To extend the nodes, PPXs need to manipulate the AST data type. Given that that data type is modified between different compiler versions, a PPX was tied to one compiler version (is that true, Carl? Or were there other workarounds before OMP?).

To gain cross-compiler compatibility, *ocaml-migrate-parsetree* (OMP)[5] was invented. It converts the compiler AST version to the version used by the PPX, then applies the PPX, and then converts back. However, downward conversions are only well-defined if the AST doesn't contain new features/nodes. So full cross-compiler compatibility is only given if

PPX AST version >= compiler AST version

That meant that the OCaml PPX world started to lag behind the compiler releases.

Furthermore, each PPX was a separate and isolated binary; with one exception: for type derivers, there was *ppx_deriving*, which would orchestrate the different derivers.

In 2018, *ppxlib*[9] was invented to solve both problems and bring more benefits (pointed out below). It exposes one fixed AST, which all OCaml PPXs are defined against. It coordinates with the compiler releases to bump that AST version in parallel. That's how it has consolidated a PPX ecosystem, which stays up-to-date.

The main problem in terms of the consolidated ecosystem is that bumping the AST breaks a few PPXs each time. To solve that situation, there was the idea to abstract the AST exposed by *ppxlib* and keep that one stable. That idea was presented at the OCaml workshop 2019[7]. However, the idea involved too much complexity and was abandoned. We now have other means to approach the stability problem, pointed out below.

2.3 Typedtree based

There have also been efforts to introduce typedtree based meta-programming: *MetaOCaml*[4]. It's been a very valuable experience and might still become more some day.

3 Current Situation

The standard for meta-programming, *ppxlib*, provides very powerful machinery. However, the most common and usually most recommended workflow is to write context-free transformations and to register those transformations via *ppxlib*'s library to *ppxlib*'s driver. The driver is one binary that orchestrates all registered transformations (PPXs). A context-free transformation locally transforms one node, isolated from its semantic context.

(visualization)

Context-free transformations have three big advantages:

1. Among all context-free transformations, we have clear composition semantics: *ppxlib*'s driver merges all context-free transformations into one single AST traversal and applies each transformation successively.
2. The just mentioned merge considerably improves performance.
3. A context-free transformation is easy to reason about and doesn't mess with its context.

3.1 Hygiene

Point 3. above touches on an important topic: PPXs generate the final representation of the program and so have an impact on how the compiler and the editor tool *merlin* analyze the program. That's why we define clear hygienic rules[6] and try to create a culture in which PPX authors are aware of the possible consequences.

Error handling

It's important to handle PPX errors at the level of the AST abstraction. Concretely, errors need to be embedded as nodes into the AST. A raising PPX means that the compiler and *merlin* don't receive an expanded AST.

Location invariants

The *Location* of a AST node reveals which part of the source code the node corresponds to. There are two location invariants, which PPXs need to respect:

- Parent-child nodes are nested wrt. locations.
- The locations of sibling nodes don't overlap.

Full qualification of identifiers and operators

A PPX needs to be independent of the semantic context it's applied in.

Example: Non-qualified identifier

Suppose you have a PPX which injects code containing

```
compare x y
```

Then the result of that comparison depends on whether *Stdlib*'s *compare* is shadowed inside the context the PPX is applied in or not.

Example: Fully qualified identifier

Now suppose your PPX instead fully qualifies:

```
Stdlib.compare x y
```

Then the result is deterministic (we assume people don't shadow the *Stdlib*).

If your PPX relies on values/modules/operators that don't form part of the *Stdlib*, you can make use of PPX runtime modules.

3.2 Stability

Additionally to being hygienic, we also want our PPX ecosystem to stay consolidated and up-to-date. The problem: when a new compiler with its new AST version is released and *ppxlib* bumps its exposed AST to the new version, there are a few PPXs that break. We have two strategies here.

Reduce the number of breakages

There are different ways to (de-)construct the AST. A very low-level approach is to handle the original data type directly. That workflow is clearly unstable by the unstable AST nature. A very high-level approach is to use the PPX *metaquot*, which comes bundled with *ppxlib*. It lets you (de-)construct nodes by writing the corresponding OCaml syntax. That's stable due to the OCaml syntax being backward compatible. An intermediate approach is to use helper modules called *Ast_builder* and *Ast_pattern*. That used to be unstable since the modules are automatically generated. Good news: since *ppxlib*.0.26.0 (2022), we keep them manually stable!

Patch each PPX in case of breakage

Even with that, there are still a few PPXs that break. Therefore, when bumping the AST, we create a big workspace containing all PPXs released on *opam* (fulfilling a few standards), called the *ppx_universe*[8]. We then quickly patch the broken PPXs in the universe and open PRs. The few affected PPX maintainers only need to review, merge and release.

References

[1] Attributes. https://ocaml.org/manual/attributes.html.

[2] Camlp4. https://en.wikipedia.org/wiki/Camlp4.

[3] Extension nodes. https://ocaml.org/manual/extensionnodes.html.

[4] Metaocaml. https://okmij.org/ftp/ML/MetaOCaml.html.

[5] Omp. https://github.com/ocaml-ppx/ocaml-migrate-parsetree.

[6] Ppx hygiene. https://ocaml.org/p/ppxlib/latest/doc/good-practices.html.

[7] Ppx in ocaml workshop 2019. https://icfp19.sigplan.org/details/ocaml-2019-papers/8/The-future-of-OCaml-PPX-towards-a-unified-and-more-robust-ecosystem.

[8] Ppx universe. https://github.com/ocaml-ppx/ppx_universe.

[9] Ppxlib. https://github.com/ocaml-ppx/ppxlib.