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

Nine years ago, the OCaml community let go of the "maintenance hell" of source code meta-programming. Now meta-programming is done on the parsetree and has evolved into a stable ecosystem. Two highlights of that evolvement: introducing parsetree migrations to gain cross-compiler-version 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 evolution of this ecosystem and its remaining challenges: guidelines for hygienic meta-programming, and stability across compiler updates.

1 Introduction

Meta-programming opens the door to boiler-plate generation, conditional compilation, domain-specific syntax extensions, and 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.

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

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 on *opam* depend 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:

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

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. Meta-programming tools, called PPXs, extend those nodes to other nodes:

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

To extend the nodes, PPXs need to manipulate the AST data type. Changes to the AST data type

would often break PPXes that depended on a previous form.

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

Initially, each PPX was a separate binary. The one early exception was *ppx_deriving*, which orchestrates multiple type-based derivers into a single rewriter.

In 2018, *ppxlib* was created, in part to solve these problems. It exposes a single version of the AST, which is kept up to date with the latest compiler. This consolidates the PPX ecosystem and upgrades most PPXes "for free".

The primary remaining problem is the few PPXes that get broken each time the AST changes. Our initial proposal, presented at the OCaml workshop 2019, was to create a single AST that could contain fragments from multiple versions at once. This proved too complex and was subsequently abandoned. We now approach stability with a combination of technical and social solutions.

2.3 Typedtree based

There have also been efforts to introduce typedtree based meta-programming, especially *MetaOCaml*. MetaOCaml and PPXes are different notions of metaprogramming with different goals and expressivity. MetaOCaml is type-safe, staged metaprogramming that can be used to specialize values at compile-time or run-time; it is built on a fork of the OCaml compiler. PPXes offer more general code generation than specializing values, but lack MetaOCaml's ability to evaluate the code and its integration with the type system.

3 Current Situation

PPX machinery and the *ppxlib* library have become the standard tool for OCaml meta-programming. They allow arbitrary AST rewriting in general, and streamlined special cases for context-free rewriting at extension nodes as well as derived code at types and module type definitions. The deriver links together multiple rewriters into one binary, executing all derivers and context-free rewriting in a single pass over an AST.

Context-free transformations have three big advantages over more general rewriting:

1. They compose well together.
2. Running in a single pass over a file is a significant performance win.
3. They are easier to reason-about than transformations that might affect, or be affected by, their 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* interpret the program. Over time we have developed hygiene guidelines for creating PPXes that "play nicely" within the ecosystem.

Error handling

Exceptions abort the transformation of a program. This leads to tools like *merlin* having no output program to analyze. As a result, we have changed *ppxlib* to catch exceptions during rewriting, and render their message as an `ocaml.error` node in the final AST. We encourage PPX authors to express errors as AST nodes directly so that they can occur in the most specific possible location, and allow the rest of rewriting to proceed.

Location invariants

The location annotated on an AST node is used by tools like *merlin* to associate the final AST with source code. This can get confusing when locations are duplicated during transformation. Our hygiene guidelines come in two parts: invariants to ensure

tools can find a unique node for a location, and tools for satifying the invariants.

Every AST node's location must contain the locations of its child nodes. Sibling nodes' locations must not overlap. These are the invariants. To help satisfy them, a location may be annotated as a "ghost" when it is a duplicate. Ghost locations are ignored for purposes of invariants, and for tools like *merlin* when performing a location-based lookup.

Full qualification of identifiers and operators

A PPX needs to be independent of the semantic context it's applied in. This property has no enforcement; it is the job of PPX authors to observe it so that their transformations are robust.

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 definitions that don't form part of the `Stdlib`, the standard practice is to mint a runtime module for your PPX, give it an explicit name such as `Ppx_example_runtime_lib`.

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*. 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.