Experiments in formatting

Frédéric Bour Hugo Heuzard Guillaume Petiot Thomas Refis

OCamlformat, the formatting tool for OCaml, could be called a (happy) accident of history: its original purpose wasn't to format OCaml source code, but to convert a ReasonML codebase to an OCaml one. It later evolved into a tool used to normalize the syntactic and layout aspects of OCaml codebases, becoming the OCamlformat that we know.

We believe that a tool dedicated to formatting OCaml source code would have a design which differs in significant ways from the one of OCamlformat. In this talk we will consider, and hopefully justify, some of these differences.

Syntax trees: Concrete vs Abstract

OCamlformat's original design was about pretty-printing OCaml source code from an OCaml AST (the Parsetree). However, the Parsetree is a partially desugared representation of OCaml source code: some constructions can be expressed with different syntaxes but the particular syntax chosen by the user is lost in the AST. For instance `let open A in e` and `A.(e)` both result in the same AST, likewise for `foo.(x)` and `Array.get foo x`, and several other such examples.

Since our goal is to format the programmer's sources, it seems reasonable to use a representation that preserves all the syntactic choices made by the user. Ranging from the choices illustrated above all the way to the presence (or absence) of parentheses. We call such a representation a Concrete Syntax Tree (or CST).

Targeting a CST implies that, unlike OCamlformat, we can't just use compiler-libs and OCaml's official parser. We need our own parser. Which begs the question: how do we make sure that we recognise the same language as the upstream parser? One way to do that is to start from the official parser and only change the semantic actions to produce our CST instead of the Parsetree. And to ensure that it's indeed all we do, we can leverage some functionality offered by menhir: given the `--only-preprocess` flag, menhir will output the grammar corresponding to a `.mly` file. So if we call menhir with that flag on our parser and on the compiler's one, we can then diff the grammars to check that they are the same.

Scope of the Experiment

One important use case of OCamlformat, which also lives under the "formatting" umbrella, is the syntactic normalisation of a codebase. For instance, with the right set of flags, OCamlformat can turn all your `begin ... end` into `(...)`, force (or remove) parentheses around tuples, etc.

With a precise enough CST, this sort of normalisation can be expressed as a CST to CST transformation. So, for now, we restrict ourselves to changing the layout of the code: we only add, or remove, whitespace.

In the absence of syntactic transformations prior to printing, a formatting tool should have the following property:

Property 1. *all the tokens in the input appear in the output, in the same order.*

Handling of comments

Using a CST as input for our formatting code gives us information about the structure of the source, thus guiding our decision with regard to the layout. However, comments in OCaml don't fit in the CST and that is because they can appear at seemingly arbitrary positions, see for instance:

- `if b1 (* && b2 *) then ()`

-
  ```
  ListLabels.map
    ~f:(* Fun.id *) (fun x (* y *) -> x)
    [ (* 3; *) 4; 5 ]
  ```
- `Base.Int.(* Hex. *)to_string 16`

When writing a formatting tool, we should aim at moving comments as little as possible when reformatting, whilst keeping the formatter code nice and focused. In particular we want to avoid having to explicitly insert some ad-hoc logic for printing comments throughout the CST printing code. This approach would be problematic because it imposes a trade off between the two goals we have. Indeed, if you try to move comments as little as possible then you need to sprinkle your comments printing logic throughout your formatter's code, obfuscating the CST printing logic. On the other hand, if you try to keep the code relatively free of this noise, then comments are going to be moved a lot in the output as they are going to be gathered around your rare insertion points.

To avoid this issue, we are going to rely on the PPrint library. Roughly speaking, this pretty printing library lets you construct the document you want to print by concatenating smaller documents. In the context of pretty printing OCaml code, that means building an atomic document (i.e. a string) for each of the tokens in the source and then building the final document representing our formatted source code by concatenating these atomic documents, interspersed with the appropriate space and line-break documents.

This model helps us because comments do not, in fact, appear at arbitrary positions. They appear in between tokens. So by writing a small wrapper around PPrint's concatenation operator, we can arrange so that comments are *implicitly* inserted when concatenating our documents.

By doing that, we get to write a naive pretty-printing code which only cares about the CST and ignores the question of comments: it is handled for us by the pretting printing engine. And we also get the following property:

Property 2. *a comment placed between two tokens in the input will be located between the same two tokens in the output.*

A property that OCamlformat does not have, and is in particular broken on the 3rd of our examples above, which OCamlformat prints as:

```
    Base.Int.to_string (* Hex. *) 16
```

Ensuring correctness

As discussed, we already make sure that we accept the same language as the compiler. But that's not enough to ensure correctness: our formatting code could forget to print some chunk of the input, or it could reorder things.

A first, and most important, correctness criterion is the following: formatting should not change the meaning of the code being formatted. Correctness is therefore implied by Property 1, which we could verify by comparing the token stream before formatting, to the one we obtain after formatting. However, Property 1 is only valid as long as the tool is restricted to changing whitespace. So we decided to implement a weaker but more robust and "future-proof" check that validates this correctness criterion by parsing both the input and the output with the compiler's parser and diffing the resulting parsetrees: they should be the same.

An additional correctness criterion, which is not enforced by the previous check and had to be implemented separately, is that every comment appearing in the input should also appear in the output, in the same order. This, again, is weaker than Property 2, which could only be enforced by checking the token stream, but was prefered for the same reason as before.

Conclusion and future work

Aiming to produce a formatter of OCaml source code gives birth to a design which differs in significant ways from the one initially chosen for OCamlformat:

- the use of a CST instead of an AST; allowing a separation of concerns between normalization and layout
- the use of a concatenative document model for the printing; making the insertion of comments implicit

We find that these particular design decisions make for a cleaner and simpler code, which we hope will be easier to maintain. Some of the design choices presented here are already making their way into OCamlformat, the use of a CST in particular, and we are currently considering how best to integrate the rest.

Another part of the design space we have yet to explore is how to handle syntactically invalid inputs. We believe that the concatenative approach to documents could help us here too. Assume a syntactic recovery mechanism, in the spirit of Merlin, which returns some CST as well as all the tokens present in the source that weren't used to build this CST. We could keep the existing code for printing the CST, and update our PPrint wrapper so that remaining tokens are inserted implicitely, in the same way as it currently inserts comments. These tokens could themselves be formatted using a different algorithm, akin to what ocp-indent does, where indentation changes are guided by the appearance of specific tokens, rather than by the syntactic structure of the input.