GopCaml: A Structural Editor for OCaml

Abstract

This talk presents GopCaml-mode, the first structural editing plugin for OCaml. We will give a tour of the main plugin features, discussing the plugin's internal design and its integration with existing OCaml and GNU Emacs toolchains.

ACM Reference Format:

. 2021. GopCaml: A Structural Editor for OCaml. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages.

1 Introduction

Language-aware editor support can vastly improve the overall user experience of a programming language. In this talk, we focus on the task of providing *syntactic* editor support for OCaml, presenting GopCaml-mode, a plugin for GNU Emacs, that extends Emacs with support for *syntax-directed* editing operations. As it turns out, the particular style of OCaml syntax produces unique challenges for capturing the structure of programs within the interfaces of standard editors.

Consider the task of providing structural editing support for GNU Emacs, one of the most popular editors within the OCaml community [1]. The base interface of Emacs revolves around a user interacting with a file by expressing operations in terms of character-based transformations of text (*i.e.* insert character at cursor). Plugins then provide additional operations to allow a user to express *syntax*-based transformations of text in terms of these lexical operations, *i.e.* delete expression at cursor *delimited by braces*, swap statements at cursor *separated by semicolons*. In the case of OCaml, this approach falls short, as OCaml's syntax means that the structure of OCaml programs can not consistently be approximated from a lexical analysis alone — what denotes the start and end of a given expression?

While OCaml has a rich tooling ecosystem, existing plugins do not adequately provide support for structural editing. Tuareg, Vim-OCaml and VS-code-OCaml-platform, are the main plugins that provide OCaml-specific language support for the popular editors, GNU Emacs, Vim and VS-Code respectively. While all three plugins provide basic structural editing support, these functionalities are implemented using basic lexical analysis alone, and so, by design, are not intended to provide a fully accurate encoding of OCaml syntax, which results in their inability to support common OCaml code transformations, such as swapping branches of a match-statement. The Merlin [2] language server extends an editor to provide additional *semantic* integration with the language, but does not focus on lower-level *syntactic* editing support. More recently, the Rotor [4–6] tool is a framework designed for refactoring OCaml projects, however its focus is on large-scale whole program transformations rather than local syntactic transformations as needed for structural editing.

Conference'17, July 2017, Washington, DC, USA

2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00

2 A tour of GopCaml

In this section, we highlight a selection of structural editing operations on OCaml mode that GopCaml-mode supports.

Consider the following snippet of code, where the position of the user's cursor is denoted by the hollow caret block:

```
let rec map f xs = match xs with
  | [] -> []
  | x :: xs -> f x :: map f xs
```

Now, with this code in hand, what can GopCaml-mode do?

Structural Navigation. One useful operation when editing code is to move the cursor relative to the structure of the program, *i.e.* move to the start of the match statement *enclosing* the cursor. GopCaml-mode explicitly supports such movements — in this case, through a function `structural-up`, which uses the concrete syntax tree (CST) for the current program to reposition the cursor accordingly:

```
let rec map f xs = match xs with
  | [] -> []
  | x :: xs -> f x :: map f xs
```

Structural Transposition. Another useful way in which syntactic information can be of use for editing is through structural transformations, such as swapping the nearest syntactic constructs by the cursor. Again, GopCaml-mode supports such operations — in this case, through a function `structural-transpose`, which uses the concrete syntax tree for the current program to modify the program text as follows:

```
let rec map f xs = match xs with
  | x :: xs -> f x :: map f xs
  | [] -> []
```

Structural Deletion. As a final example, consider the task of deleting entire nodes from the syntax tree, *i.e.* delete the branch at the cursor. GopCaml-mode supports this transformation through a function `structural-delete`, which again relies on the concrete syntax tree for the current program to remove the nearest syntax construct to the cursor, leaving the buffer as follows:

```
let rec map f xs = match xs with
  | x :: xs -> f x :: map f xs
```

Other operations supported by GopCaml-mode include:

- **Structural selection** - select regions using the CST.
- **Structural syntax-move** - move nodes around the CST.
- **Extract expression** - extract a common sub-expression to a let binding.
- **Jump to binding/parameter** - move the cursor to the nearest let binding/parameter.

3 Under the hood

We now present the core logic used by GopCaml-mode to provide structural editing — a variant of Huet's zipper [3], specialised for navigating syntax trees. A syntax tree might provide the information needed for structural editing, but it is fundamentally unsuited for use in an interactive setting where the tree traversal is iterative, typically being controlled by the user. The definition of the zipper used in GopCaml-mode is as follows:

```
type zipper =
  | Top
  | Node of {
      item: t; below: t list; above: t list;
      parent: zipper;
      bounds: text_region;
    }
```

An instance of this zipper encodes a path from the root of the program CST (*i.e.* `Top`) to a currently *focused* node (`item`), retaining the structure of the whole tree by tracking the siblings below and above each node.

The main modification made to Huet's original definition is to extend each intermediate node with an additional field that captures the "bounds" of the current node — the character range in the original buffer from the start to the end of the current node. This change enables a simple interface between the core logic and the user's editor — that of simple text ranges, as will be discussed later in this section.

Finally, the type `t` captures a thin wrapper around OCaml's underlying AST type:

```
type t =
  | Sequence of
      text_region option * t list * t * t list
  | Signature_item of Parsetree.signature_item
  (* ... *)
```

The full data type has a constructor for each node in the OCaml AST, along with an additional `Sequence` constructor which provides a generic encoding of the "minimal" information needed to represent an syntactic node — this happens to be particularly useful for handling intermediate operations that produce invalid syntax trees.

As it turns out, using a zipper to track the user's position in the syntax tree allows for a rather simple and elegant implementation of the structural operations we saw earlier:

- **Structural movement** - With this framework, syntax-based movement simply boils down to moving the zipper itself and then updating the position of the editor's cursor to reflect the change in the focus of the zipper. For example, in order to move the cursor to the enclosing match osition expression as before, the plugin first moves the zipper up the syntax tree by replacing it with the value of its parent and then simply updates the editor's cursor to move to the start of the newly focused element.
- **Structural transposition** - Syntax-based transformations also neatly fit into this framework. For example, in order to transpose the match branches from before, the plugin first retrieves the two nodes to be changed as the focused element of the zipper and its next sibling (from the `item` and `above` fields respectively). After swapping them in the zipper, to perform this transformation on the program text itself, the plugin passes the text ranges corresponding to the swapped elements to the editor, which then simultaneously swaps the corresponding characters.
- **Structural deletion** - Structural deletion requires more care, but still works quite cleanly with zippers. In order to delete an element in the zipper, the plugin need only remove the currently focused item and replace it with its next sibling, and similarly ask the editor to delete the text region corresponding to the removed element. Finally, to account for the removed characters from the text buffer, the zipper must update the text bounds for all subsequent and parent nodes accordingly — by shifting or shrinking them respectively.

4 Editor integration and Future Work

We have implemented the above zipper-based structural editing framework as a small OCaml library, using the OCaml compiler infrastructure to provide the AST definitions and parser. The GopCaml-mode Emacs plugin builds on this core framework to integrate it with GNU Emacs, providing functions to persist and cache the state of the zipper over an editing session and keybindings to run structural operations using the zipper. For example, when the user presses a keybinding for a structural operation, GopCaml-mode then transparently handles the tasks of building the CST, constructing a zipper at the cursor and performing the editing operation on the zipper and text buffer.

In the future, we plan to extend support to other editors. The core editing framework is agnostic to the particular choice of editor, simply implementing edits in terms of zipper transformations as above, so it would not be too challenging to integrate this into editors such as Vim and VS-code. Finally, looking further out, we hope to extend this framework further, moving from the *syntactic* to the *semantic* - for instance, using typing information to guide transformations.

References

[1] 2020. OCaml User Survey 2020. https://docs.google.com/forms/d/1OZV7WCprDnouU-rIEuw-1lDTeXrH_naVlJ77ziXQJfg/viewanalytics [Online; accessed 19. May 2021].

[2] Frédéric Bour, Thomas Refis, and Gabriel Scherer. 2018. Merlin: a language server for OCaml (experience report). *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–15.

[3] Gérard Huet. 1997. The zipper. *Journal of functional programming* 7, 5 (1997), 549–554.

[4] Reuben NS Rowe, Hugo Férée, Simon J Thompson, and Scott Owens. 2019. Rotor: a tool for renaming values in OCaml's module system. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*. IEEE, 27–30.

[5] Reuben NS Rowe and Simon J Thompson. 2017. Rotor: First steps towards a refactoring tool for OCaml. In *OCaml Users and Developers Workshop 2017*.

[6] Reuben NS Rowe and Simon J Thompson. 2018. Towards Large-scale Refactoring for OCaml. *Proceedings of the ACM on Programming Languages* 1, 1 (2018).