Commanding Emacs from Coq

An editor command eDSL in Coq, interpreted in Emacs Lisp

JOOMY KORKUT, Princeton University, USA

1 MOTIVATION AND DESCRIPTION

Interactive theorem provers Coq, Agda and Idris allow programmers to ask questions about incomplete programs (or proofs) to Emacs, and get meaningful answers based on types of the missing parts. This turns programming into a conversation between programmers and their compilers, where Emacs acts as an intermediary. In Agda and Idris these questions are asked through editor actions usually involving holes in the program. Coq's interactivity model, however, is fundamentally different; it is based on stepping through proofs. Specifically, Coq users step through vernacular commands and tactics. Vernacular commands are the primary way to interact with the program as a whole. They can check the types of terms or definitions, add new definitions, load new files, open and define modules and do many other tasks. Coq users can even ask Coq to ensure that a given vernacular command will **`Fail`**, or they can **`Time`** how long a command will take.

In all of the aforementioned languages, programmers are limited in the kinds of questions they can ask. Recent work on extensible editor actions in Idris [1] and Hazel [2, 3] leverages metaprogramming techniques to specify how the compiler should answer custom questions, but that requires writing code in the editor language (Emacs Lisp etc.) to glue the compiler and the editor interface because metaprograms can only command the compiler and not the editor itself.

Coq has a different story about its ability to answer new kinds of questions, since users can define new vernacular commands in ML plugins. However, even vernacular commands cannot achieve rich editor actions that involve changing text in the buffer, they can only print information on the screen. Furthermore, it is impossible to define new vernacular commands in Coq itself.

We will describe *editor metaprogramming*, a novel idea of writing metaprograms that command the editor. We define a monadic embedded domain-specific language (eDSL) for editor commands in Coq, which enables composing these commands in meaningful ways, with the full power of Coq! We write an interpreter for this eDSL in Emacs Lisp, which will execute one atomic editor action, such as deleting a character or moving the cursor, and send the rest of the computation back to Coq for evaluation, which then results in Coq finding another atomic editor action that Emacs Lisp must execute. Emacs and Coq processes will communicate back and forth until the entire computation is completed.

The simplest (and minimized) form of this eDSL is an inductive data type as defined below:

```
Inductive edit : Type -> Type :=
| ret : forall {a}, a -> edit a
| bind : forall {a b}, edit a -> (a -> edit b) -> edit b
| insert_char : ascii -> edit unit
| remove_char : edit unit
| get_char : edit ascii
| move_left : edit unit
| move_right : edit unit.
```

Author's address: Joomy Korkut, Princeton University, Princeton, New Jersey, USA, joomy@cs.princeton.edu.

Submitted to the Coq Workshop 2019, at ITP 2019 in Portland, Oregon, USA.

Each atomic editor action we want to have is a constructor in this data type. For example, `insert_char` inserts a given `ascii` character to the buffer right before the cursor, `remove_char` deletes one character in the buffer right after the cursor, and `get_char` gives you the character under the cursor.

The `ret` and `bind` constructors are a common trick that is used to design monadic eDSLs quickly. Once we have them, defining a `Monad` type class instance of the `edit` type is straightforward:

```
Instance Monad_edit : Monad edit := {ret a := @ret a ; bind a b := @bind a b}.
```

Now that have a `Monad` instance for `edit`, we can compose editor actions to make larger building blocks. For instance, let's build an editor action that replaces the character under the cursor:

```
Definition replace_char (c : ascii) : edit unit := remove_char ;; insert_char c.
```

Using this, we can now define an editor action that takes the character under the cursor, shifts is ASCII value by one, and replaces it with the same character:

```
Definition shift : edit unit :=
  c <- get_char ;; replace_char (ascii_of_N (1 + N_of_ascii c)) ;; move_right.
```

Now you can evaluate the Emacs Lisp command `(run "shift")` and observe that it indeed replaces the character under the cursor and moves the cursor. h`e`llo should become hf`l`lo.

2 IMPLEMENTATION

We need to write an interpreter in Emacs Lisp that looks up the definition of the required editor action, executes the first atomic editor action, and sends the result of the atomic action and the rest of the computation back to Coq. This is exactly what our `run` function in Emacs Lisp does. Using Proof General's internals, it sends a vernacular command to Coq, of the form `Eval cbn in (...)`. In order to keep the interpreter in Emacs Lisp as simple as possible, we can make sure that the term sent to Emacs has an atomic editor action right in the beginning; the action must be atomic itself, or must have the form `bind <atomic> (fun x => ...)`.

At this point, associativity of bind comes to our help. We know that `bind (bind m f) g` is equivalent to `bind m (fun x => bind (f x) g)`. If we repeatedly apply this transformation to composed editor actions, that would make all editor actions fit the form we require in our interpreter in Emacs Lisp. The function that does the right association transformation is written in Coq; we can call it `right_assoc`.

Therefore, the vernacular command run by the our Emacs Lisp function called `run` is

```
Eval cbn in (right_assoc shift).
```

Our interpreter then parses the response from Coq, which would be of the form

```
= bind get_char (fun x => bind remove_char (fun _ => ...)) : edit unit
```

The interpreter in Emacs Lisp will detect that the first atomic action is `get_char`, then get the character under the cursor, and create a string that represents the term `(fun x => bind remove_char (fun _ => ...)) "e"`, and send that back to Coq with the same vernacular command. This back and forth communication will continue until the response received from Coq only contains a single atomic action and no `bind`s.

Our prototype implementation is available at http://github.com/joom/metaprogrammable-editor. While this implementation is brittle and lacking in helpful error messages, it shows the capabilities of such a mechanism.

REFERENCES

[1] Joomy Korkut and David Thrane Christiansen. 2018. Extensible Type-Directing Editing. In *The 3rd ACM SIGPLAN International Workshop on Type-Driven Development*.

[2] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. In *Summit on Advances in Programming Languagess (SNAPL) (LIPIcs)*, Vol. 71. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 11:1–11:12.

[3] Xuanrui (Ray) Qi. 2019. From Tactics to Structure Editors for Proofs. In *Proceedings of Off the Beaten Track 2019 (OBT19)*.