Commanding Emacs from Coq

(title: Emacs Lisp considered harmful)

Joomy Korkut
Princeton University

Scheme Workshop, August 18th 2019
Coq

• Interactive theorem prover with similar syntax to OCaml.

• Has amazing Emacs support, thanks to Proof General.
Definition $b := \text{andb true false}$. 

Check $b$. 

Eval compute in $b$. 
Definition \( b := \text{andb true false}. \)

Check \( b \).

Eval compute in \( b \).
Definition $b := \text{andb} \; \text{true} \; \text{false}$.

Check $b$.

Eval compute in $b$.  

b : bool
Definition \( b := \text{andb true false.} \)

Check \( b \).

Eval compute in \( b \).

\[
\begin{align*}
= \ & \text{false} \\
: \ & \text{bool}
\end{align*}
\]
In the background...

Emacs -> Proof General -> Coq

request as vernacular command

response as output
Here's what I want to do
Definition to_upper
   : ascii → ascii :=
   ...

Definition make_upper
   : edit unit :=
   do c ← get_char ;;
   replace_char (to_upper c).
Definition to_upper
  : ascii -> ascii :=
  ...

Definition make_upper
  : edit unit :=
  do c <- get_char ;;
  replace_char (to_upper c).

joomy
  entering new text into buffer (cursor in the beginning)
Definition **to_upper**

: ascii \(\rightarrow\) ascii :=

...  

Definition **make_upper**

: edit unit :=

  do c \(\leftarrow\) get_char \(\;;\) \(\;;\)
    replace_char (to_upper c).

joomy

M-: (run "make_upper") RET  

We run some Emacs command
**Definition** to_upper

: ascii \(\rightarrow\) ascii :=

... 

**Definition** make_upper

: edit unit :=

  do c \(\leftarrow\) get_char ;;
  replace_char (to_upper c).

---

**Joomy**

The character under the cursor is changed into uppercase!
What did we do here?

• We defined an editor macro in Coq.

• This macro depends on the computation of nontrivial Coq functions.

• We ran this editor macro in Emacs Lisp.
How did we do that?

• We defined an embedded domain-specific language (eDSL) in Coq, that helps users define editor macros.

• We wrote an interpreter for this Coq eDSL in Emacs Lisp.

• This interpreter **executes the atomic actions in Emacs**.

• Whenever the interpreter sees an uncomputed expression, it sends the expression back to Coq for call-by-need evaluation!
Let's illustrate that. Remember the macro we ran?

```plaintext
Definition to_upper
    : ascii → ascii :=
    ...

Definition make_upper
    : edit unit :=
    do c ← get_char ;;;
    replace_char (to_upper c).
```
Tracing our steps

Emacs

```
get_char MN= \( fun \ c \Rightarrow \ldots \)"
```

```
remove_char MN= \( fun \ _\Rightarrow \ldots \)"
```

```
insert_char "J"
```

Emacs now realizes that the macro execution is complete!
The definition of our eDSL

\[
\text{Inductive edit : Type \rightarrow Type :=}
\]

- ret : forall \{a\}, a \rightarrow edit a
- bind : forall \{a b\}, edit a \rightarrow (a \rightarrow edit b) \rightarrow edit b
- message : string \rightarrow edit unit
- message_box : string \rightarrow edit unit
- input : edit string
- insert_char : ascii \rightarrow edit unit
- remove_char : edit unit
- get_char : edit ascii
- move_left : edit unit
- move_right : edit unit.

Constructors except bind are called atomic.

connection with free monads?
(find me after the talk if you know more!)
The definition of our interpreter

(defun run-action (a)
  (pcase a
    `(ret ,x) x)
  `(message ,s) (message s) "tt")
  `(message-box ,s) (message-box s) "tt")
  `(insert-char ,c) (insert c) "tt")
  ('get-char (prin1-to-string (string (following-char)))))
  ('remove-char (delete-char 1) "tt")
  ('move_right (right-char) "tt")
  ('move_left (left-char) "tt")
  ('move_up (previous-line) "tt")
  ('move_down (next-line) "tt")
  ('move_beginning (move-beginning-of-line) "tt")
  ('move_end (move-end) "tt")
  (l (message "Unrecognized action") nil)))
The definition of our interpreter

(defun parse-response (s)
  (let* ((untail ...))
    (pcase (read-from-string untail)
      (= . ,m)
        (pcase ...
          (bind . ,n)
            (pcase (read-from-string (substring untail (+ m n 1)))
              (= ,act . ,p)
                (run (concat (substring untail (+ m n p 1)) " " (run-action act))))
              (message "Error: Expecting either a bind or an action."))
          (message "Error: Expecting = in the beginning of the output."))
      (l (message "Error: Expecting either a bind or an action.")))
    (l (message "Error: Expecting = in the beginning of the output."))))

(defun run (s)
  (let* ((res (proof-shell-invisible-cmd-get-result (concat "Eval cbn in (right_assoc (" s ")).")))
    (parse-response res)))
One little caveat

• We assume that the macro definition Emacs receives is either \( m \gg= f \), where \( m \) is an atomic action, or full the macro definition an atomic action itself.

• Not all macros written with our eDSL would fit this format!

• However, we can restructure a macro definition to fit this format! Since \texttt{edit} is a monad, this is just right association of monadic bind!
Right association of bind

\[(\text{get\_char} \gg= \text{insert\_char}) \gg= (\text{fun } _ \Rightarrow \text{move\_right})\]

repeat this transformation until the left hand side is atomic

We have a fuel based Coq function to do that!

\[\text{get\_char} \gg= (\text{fun } c \Rightarrow (\text{insert\_char } c) \gg= (\text{fun } _ \Rightarrow \text{move\_right}))\]
What's the end goal here?

- We can define IDE features for Coq in Coq!
  - Requires a more elaborate eDSL
  - Requires better Coq support for type-directed development