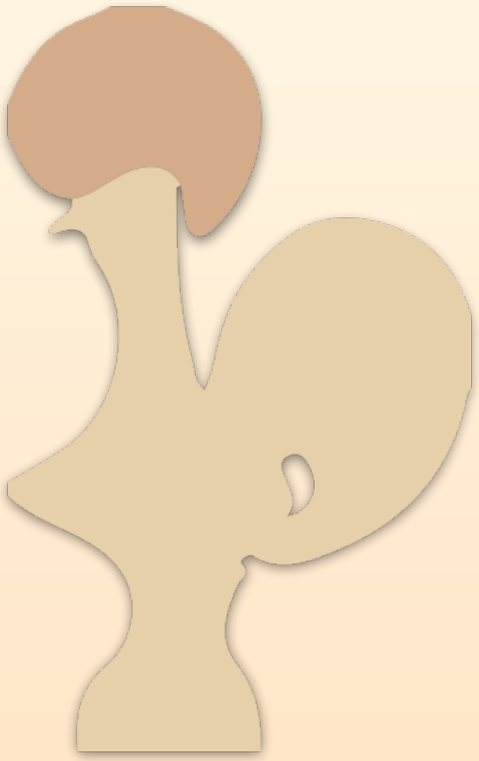
⚡ talk

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**.

Emacs

```
Definition b := andb true false.

Check b.

Eval compute in b.
```

Emacs

```
Definition b := andb true false.

Check b.

Eval compute in b.
```

stepping through

```
b is defined
```

Emacs

```
Definition b := andb true false.

Check b.

Eval compute in b.
```

```
b
     : bool
```

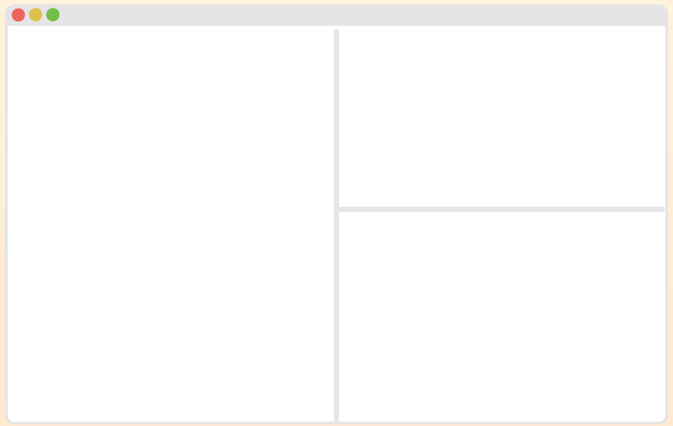
Emacs

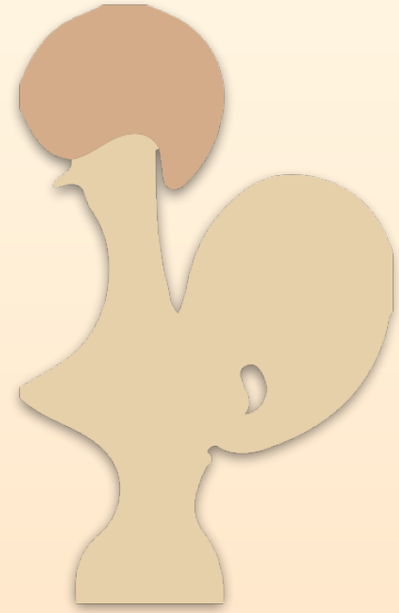
```
Definition b := andb true false.

Check b.

Eval compute in b.
```

```
= false
: bool
```

vernacular commands

In the background...





Emacs

request as vernacular command

response as output

Coq

Proof General

Here's what I want to do

Emacs

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

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

Emacs

```
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)

Emacs

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

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

joomy

M-: (run "make_upper") RET

We run some Emacs command

Emacs

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

Definition make_upper
             : edit unit :=
  do c ← 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!

Emacs

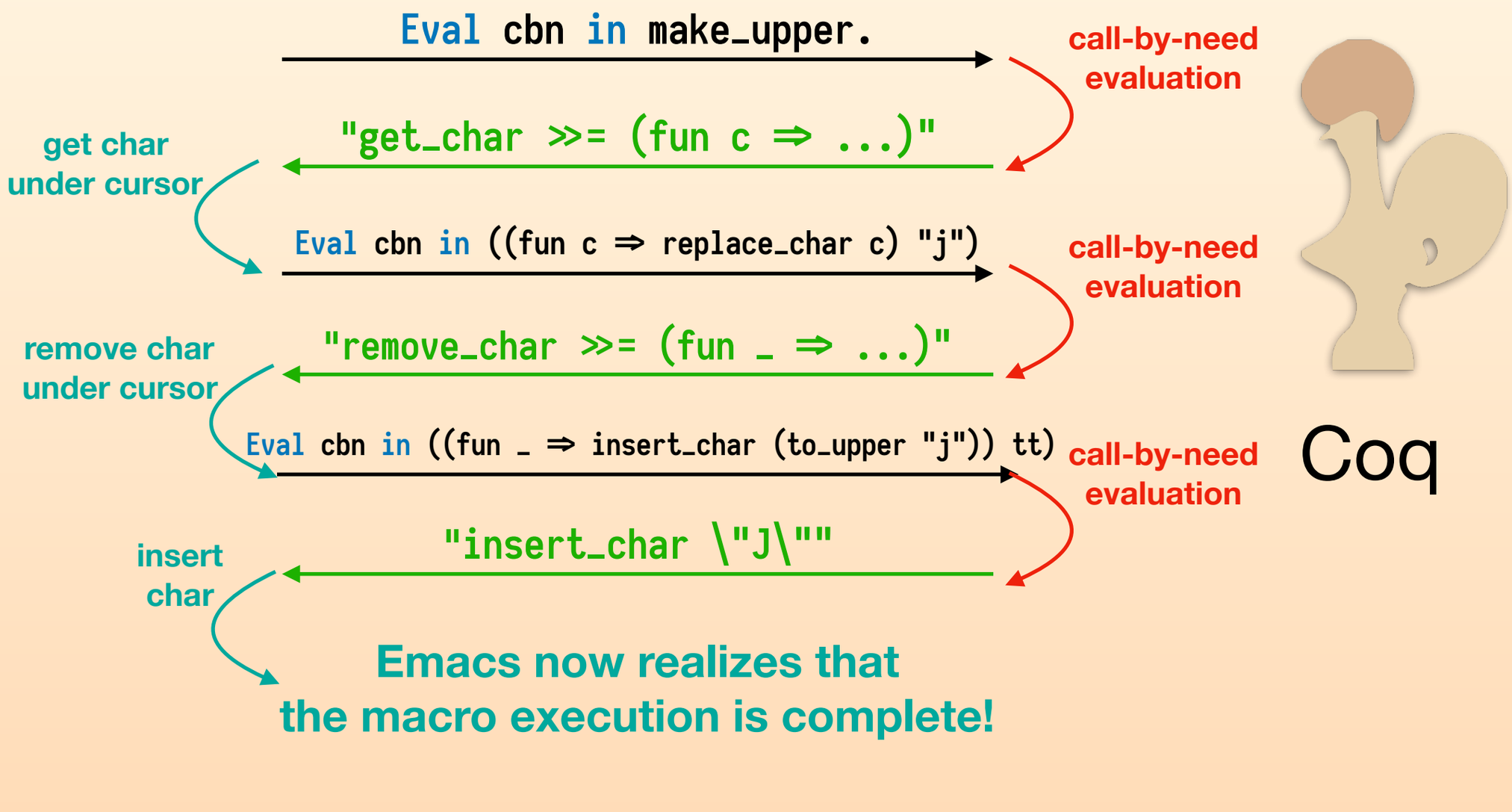
```
Definition to_upper
             : ascii → ascii :=
  ...

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

Joomy

Let's illustrate that. Remember the macro we ran?

Tracing our steps

Emacs



Coq

The definition of our eDSL

```
Inductive edit : Type → Type :=
| ret : forall {a}, a → edit a
| bind : forall {a b}, edit a → (a → edit b) → edit b
| message : string → edit unit
| message_box : string → edit unit
| input : edit string
| insert_char : ascii → edit unit
| remove_char : edit unit
| get_char : edit ascii
| move_left : edit unit
| move_right : edit unit.
```

connection with free monads?

(find me after the talk if you know more!)

Constructors except **bind** are called atomic.

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))))))
           (`(,act . ,m) (run-action act))
           (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)))
```

parsing with string operations elided here

from Proof General

One little caveat

- We assume that the macro definition Emacs receives is either `m >>= 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 **`edit`** is a monad, this is just right association of monadic bind!

Right association of bind

```
(get_char >>= insert_char) >>= (fun _ ⇒ move_right)
```

repeat this transformation
until the left hand side is atomic

We have a fuel based
Coq function to do that!

```
get_char >>= (fun c ⇒ (insert_char c) >>= (fun _ ⇒ 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