Functional Evaluation Contexts

An alternative way to handle evaluation contexts in a proof assistant like Coq

Jules Jacobs

September 22, 2021

1 Introduction

One usually starts a language definition with a BNF grammar for its syntax:

$$e ::= x \mid n \mid \lambda x.e \mid e\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid e + e$$

Then one defines the operational semantics:

$$\frac{n \neq 0}{\mathsf{if}\ n\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \to e_1} \qquad \frac{n = 0}{\mathsf{if}\ n\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \to e_2}$$

$$\frac{\cdot}{(\lambda x.e_1)e_2 \to e_1[x \mapsto e_2]} \qquad \frac{\cdot}{n + m \to [n + m]}$$

These rules only apply when the outermost expression can take a step. In order to complete the operational semantics we need evaluation contexts. They are usually defined with a BNF grammar for expressions with a hole:

$$E ::= [.] \mid E + e \mid n + E \mid \mathsf{if}\ E\ \mathsf{then}\ e\ \mathsf{else}\ e \mid E\ e$$

And a rule for stepping in a context:

$$\frac{e_1 \to e_2}{E[e_1] \to E[e_2]}$$

The syntax $E[e]$ means substituting $[.] \mapsto e$ in E. Thus, an evaluation context may be seen as a function from expressions to expressions. Without evaluation contexts, we would have to write a series of rules to step in each context:

$$\frac{e_1 \to e_2}{e_1 + e_3 \to e_2 + e_3} \qquad \frac{e_1 \to e_2}{n + e_1 \to n + e_2} \qquad \frac{e_1 \to e_2}{e_1\ e_2 \to e_2\ e_3}$$

$$\frac{e_1 \to e_2}{\mathsf{if}\ e_1\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4 \to \mathsf{if}\ e_2\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4}$$

Evaluation contexts let us write these rules as a single rule.

2 Traditional evaluation contexts in Coq

To translate such a language definition to a proof assistant like Coq, we translate:

- The BNF grammar for expressions to an expr algebraic data type.
- The head step semantics to a relation head_step : expr → expr → Prop.
- The BNF grammar for evaluation contexts to an algebraic data type ctx.
- The operation $E[e] = \mathsf{fill}\ E\ e$ using a function fill : ctx → expr → expr.
- The final step relation to step : expr → expr → Prop

For the language given previously, this would be done as follows:

```
Inductive head_step : expr → expr → Prop :=
  | Step_add : ∀ n m, head_step (Add (Nat n) (Nat m)) (Nat (n+m))
  | Step_if_t : ∀ e1 e2 n, head_step (If (Nat (S n)) e1 e2) e1
  | Step_if_f : ∀ e1 e2, head_step (If (Nat 0) e1 e2) e2
  | Step_app : ∀ e1 e2 s, head_step (App (Lam s e1) e2) (subst s e2 e1).

Inductive ctx :=
  | Ctx_Hole : ctx
  | Ctx_AddL : expr → ctx → ctx
  | Ctx_AddR : nat → ctx → ctx
  | Ctx_If : expr → expr → ctx → ctx
  | Ctx_AppL : expr → ctx → ctx.

Fixpoint fill E x :=
  match E with
  | Ctx_Hole ⟹ x
  | Ctx_AddL e2 E' ⟹ Add (fill E' x) e2
  | Ctx_AddR n E' ⟹ Add (Nat n) (fill E' x)
  | Ctx_If e1 e2 E' ⟹ If (fill E' x) e1 e2
  | Ctx_AppL e2 E' ⟹ App (fill E' x) e2
  end.

Inductive step : expr → expr → Prop :=
  | Step_ctx : ∀ E e1 e2,
      head_step e1 e2 → step (fill E e1) (fill E e2).
```

This is the style common in Iris.

3 Functional evaluation contexts

An alternative is to explicitly define evaluation contexts as a set of functions $E \subseteq \mathsf{expr} \to \mathsf{expr}$:

$$\begin{aligned} E = &\{(x \mapsto x + e) \mid e \in \mathsf{expr}\} \cup \\ &\{(x \mapsto n + x) \mid n \in \mathbb{N}\} \cup \\ &\{(x \mapsto \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \mid e_1, e_2 \in \mathsf{expr}\} \cup \\ &\{(x \mapsto x\ e) \mid e \in \mathsf{expr}\} \end{aligned}$$

Or, in BNF syntax:

$$E(x) ::= x + e \mid n + x \mid \mathsf{if}\ x\ \mathsf{then}\ e\ \mathsf{else}\ e \mid x\ e$$

The substitution $E[e] = E[[.] \mapsto e]$ in the semantics becomes ordinary function application:

$$\frac{e_1 \to e_2 \qquad k \in E}{k(e_1) \to k(e_2)}$$

This works nicely in Coq too:

```
Inductive ctx : (expr → expr) → Prop :=
  | Ctx_AddL : ∀ e2, ctx (λ x, Add x e2)
  | Ctx_AddR : ∀ n, ctx (λ x, Add (Nat n) x)
  | Ctx_If : ∀ e1 e2, ctx (λ x, If x e1 e2)
  | Ctx_AppL : ∀ e2, ctx (λ x, App x e2).

Inductive step : expr → expr → Prop :=
  | Step_add : ∀ n m, step (Add (Nat n) (Nat m)) (Nat (n+m))
  | Step_if_t : ∀ e1 e2 n, step (If (Nat (S n)) e1 e2) e1
  | Step_if_f : ∀ e1 e2, step (If (Nat 0) e1 e2) e2
  | Step_app : ∀ e1 e2 s, step (App (Lam s e1) e2) (subst s e2 e1)
  | Step_ctx : ∀ e1 e2 k, ctx k → step e1 e2 → step (k e1) (k e2).
```

This is the style common in CompCert.
This way of presenting evaluation contexts is more concise and makes it quite clear that it's equivalent to having separate stepping rules for each context.
One might worry that we run into issues due to Coq's lack of function extensionality, but it turns out that we never have to prove that two functions are equal. We only need to destruct and construct ctx's, which works perfectly fine. A Coq file that has the above language definition, a type system, and a progress and preservation proof, can be found at:

https://julesjacobs.com/notes/functionalctxs/functionalctxs.v