Record Updates in Coq

Tej Chajed

CoqPL 2021

Updating records is too hard

```
Record X := mkX { A: nat; B: nat; C: bool; }.
```

Updating records is too hard

```
Record X := mkX { A: nat; B: nat; C: bool; }.

Definition setB (f: nat -> nat) (x: X): X :=
  mkX (A x) (f (B x)) (C x).
```

Updating records is too hard

```
Record X := mkX { A: nat; B: nat; C: bool; }.

Definition setA f x :=
  mkX (f (A x)) (B x) (C x).

Definition setB f x :=
  mkX (A x) (f (B x)) (C x).

Definition setC f x :=
  mkX (A x) (B x) (f (C x)).
```

Updating records is too hard

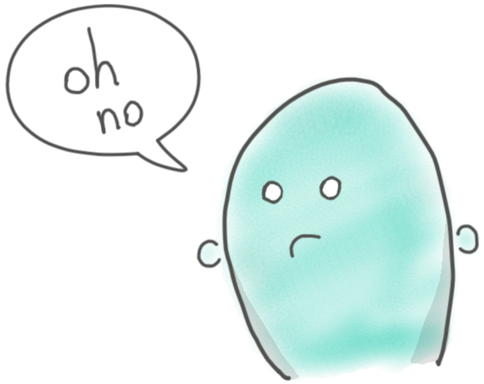
```
Record X := mkX { A: nat; B: nat; C: bool; D: list nat; }.

Definition setA f x :=
  mkX (f (A x)) (B x) (C x) (D x).

Definition setB f x :=
  mkX (A x) (f (B x)) (C x) (D x).

Definition setC f x :=
  mkX (A x) (B x) (f (C x)) (D x).

Definition setD f x :=
  mkX (A x) (B x) (C x) (f (D x)).
```

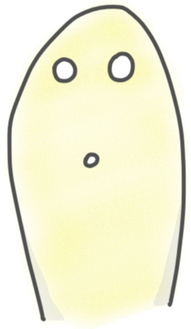


coq-record-update

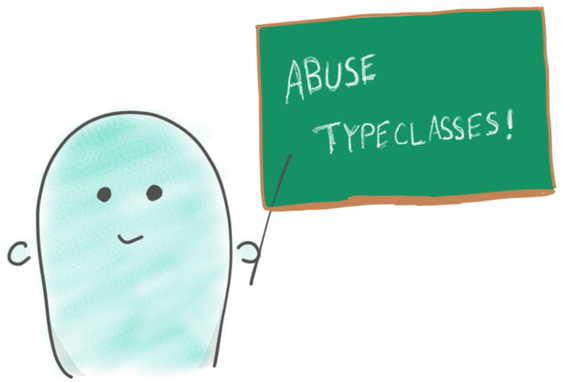
```
Record X := mkX { A: nat; B: nat; C: bool; }.

Instance etaX : Settable X := fun x => mkX (A x) (B x) (C x).

Definition setB f x := set B f x.
```



Implementing coq-record-update



Represent record fields (typeclass)

Construct setter for a field (Ltac)

Export a nice interface (typeclass hackery)

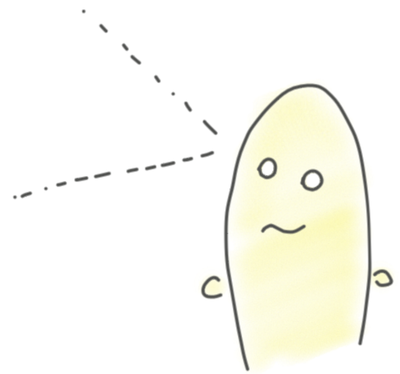
Representing a record’s fields

```
Record X := mkX { A: nat; B: nat; C: bool; }.

Definition etaX := fun x => mkX (A x) (B x) (C x).
```

Observe that etaX is a prototype for any field update, eg setB:

```
Definition setB f :=
  fun x => mkX (A x) (f (B x)) (C x).
```



Ltac implementation of `solve_setter`

```
Ltac solve_setter R proj :=
```

Ltac implementation of `solve_setter`

```
Ltac solve_setter R proj :=
  let eta := eval hnf in (_ : Settable R) in
  fun x => mkX (A x) (B x) (C x)
```

Lookup eta expansion

Ltac implementation of `solve_setter`

```
Ltac solve_setter R proj :=
  let eta := eval hnf in (_ : Settable R) in
  lazymatch (eval pattern proj in eta) with
  | ?set_f _ =>
```

Lookup eta expansion

Abstract over `proj`

```
fun x => mkX (A x) (B x) (C x)
```

```
fun (proj: X -> nat) =>
  fun x => mkX (A x) (proj x) (C x)
```

Ltac implementation of `solve_setter`

```
Ltac solve_setter R proj :=
  let eta := eval hnf in (_ : Settable R) in
  lazymatch (eval pattern proj in eta) with
  | ?set_f _ =>
    constr:(fun f => …)
  end
```

Lookup eta expansion

Abstract over `proj`

Specialize to `f ∘ B` (not shown)

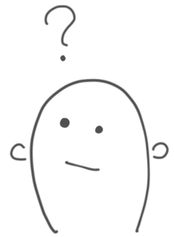
```
fun x => mkX (A x) (B x) (C x)
```

```
fun (proj: X -> nat) =>
  fun x => mkX (A x) (proj x) (C x)
```

```
fun (f: nat -> nat) =>
  fun x => mkX (A x) (f (B x)) (C x)
```

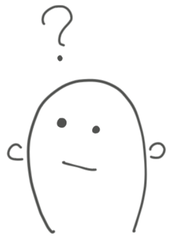
```
Class Setter {R T} (proj: R -> T) :=
  set : (T -> T) -> (R -> R).
```

What even is a Class?



```
Class Setter {R T} (proj: R -> T) :=
  set : (T -> T) -> (R -> R).
```

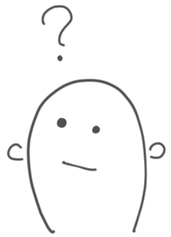
What even is a Class?



```
Class Setter {R T} (proj: R -> T) :=
  set : (T -> T) -> (R -> R).
```

```
set        B          f x
```

What even is a Class?

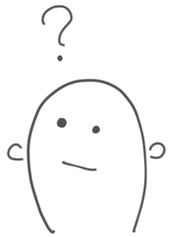


```
Class Setter {R T} (proj: R -> T) :=
  set : (T -> T) -> (R -> R).
```

```
set          B           f x
@set  X nat  B ?setter   f x
```

R and T can be inferred from proj

What even is a Class?



```
Class Setter {R T} (proj: R -> T) :=
  set : (T -> T) -> (R -> R).
```

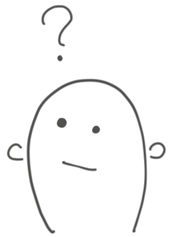
```
set          B           f x
@set  X nat  B ?setter   f x
```

R and T can be inferred from proj

Coq resolves like this:

```
⊢ ?setter : @Setter X nat B
```

What even is a Class?



```
Class Setter {R T} (proj: R -> T) :=
  set : (T -> T) -> (R -> R).
```

```
 set         B          f x
@set  X nat B ?setter f x
```

R and T can be inferred from proj

Coq resolves like this:

```
⊢ ?setter : @Setter X nat B
eauto using typeclass_instances
```

Hacking typeclasses for fun and profit

normal use of typeclasses is like this:

```
Definition setB f : Setter B :=
  fun x => mkX (A x) (f (B x)) (C x).
Hint Resolve setB : typeclass_instances.
```

Hacking typeclasses for fun and profit

normal use of typeclasses is like this:

```
Definition setB f : Setter B :=
  fun x => mkX (A x) (f (B x)) (C x).
Hint Resolve setB : typeclass_instances.
```

coq-record-update resolves Setter with Ltac instead:

```
Hint Extern 1 (@Setter ?R _ ?proj) =>
  solve_setter R proj : typeclass_instances.
```

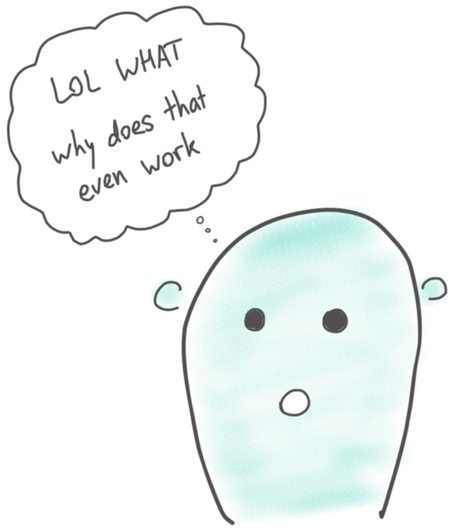
Hacking typeclasses for fun and profit

normal use of typeclasses is like this:

```
Definition setB f : Setter B :=
  fun x => mkX (A x) (f (B x)) (C x).
Hint Resolve setB : typeclass_instances.
```

coq-record-update resolves Setter with Ltac instead:

```
Hint Extern 1 (@Setter ?R _ ?proj) =>
  solve_setter R proj : typeclass_instances.
```



We can do better

see abstract

Add some sweet notation

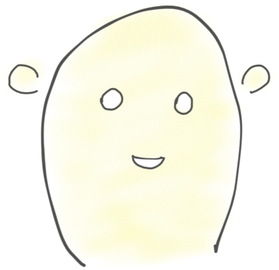
Catch errors (eg, getting order of fields wrong)

coq-record-update, once again

```
Record X := mkX { A: nat; B: nat; C: bool; }.

Instance etaX : Settable _ := settable! mkX <A; B; C>.

Definition setB b x := x <| B := b |>.
```



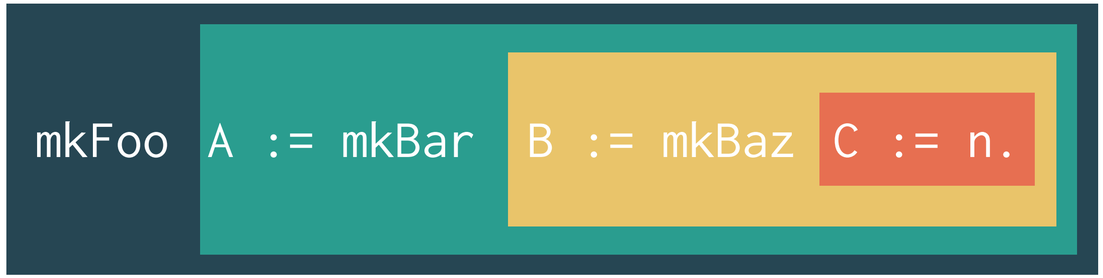
We can do even better

Use a plugin to avoid eta expansion boilerplate
(thanks to Talia Ringer)

Add notation for nested updates
(thanks to Jakob Botsch Nielsen)

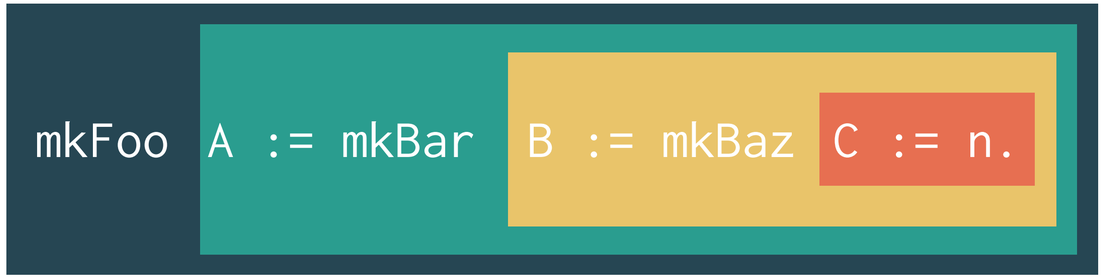
Nested updates are almost lenses

```
Record baz := mkBaz { C : nat; }.
Record bar := mkBar { B : baz; }.
Record foo := mkFoo { A : bar; }.
```



Nested updates are almost lenses

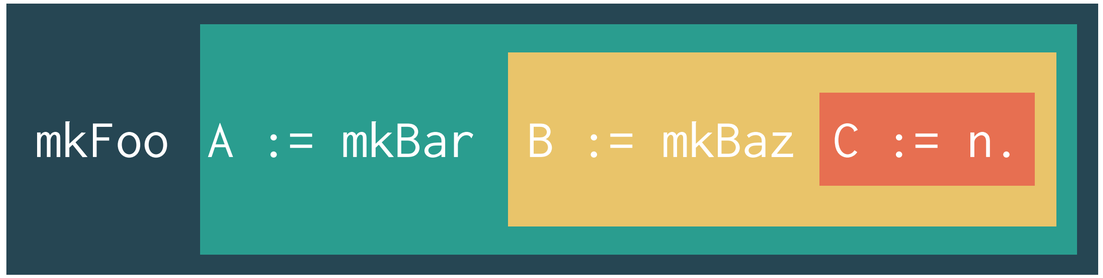
```
Record baz := mkBaz { C : nat; }.
Record bar := mkBar { B : baz; }.
Record foo := mkFoo { A : bar; }.
```




```
Instance etaBaz : Settable _ := settable! mkBaz<C>;
Instance etaBar : Settable _ := settable! mkBar<B>;
Instance etaFoo : Settable _ := settable! mkFoo<A>;
```

Nested updates are almost lenses

```
Record baz := mkBaz { C : nat; }.
Record bar := mkBar { B : baz; }.
Record foo := mkFoo { A : bar; }.
```




```
Instance etaBaz : Settable _ := settable! mkBaz<C>;
Instance etaBar : Settable _ := settable! mkBar<B>;
Instance etaFoo : Settable _ := settable! mkFoo<A>;

Definition setNested n x := x <| A; B; C := n |>.
```

coq-record-update

Simple addition of record updates to Coq

You, too, can abuse typeclasses

github.com/tchajed/coq-record-update