Record Updates in Coq

Tej Chajed
tchajed@mit.edu

Abstract

We describe the implementation of coq-record-update, a library that generates functions to set and update fields of Coq records to complement Coq's existing support for field projections. The implementation abuses features of Coq typeclasses and is thus fun to describe. The library has *industrial and academic users* that are not in the authors' institution, which lends credibility to the assertion that it is useful.

Keywords: records, typeclasses

ACM Reference Format:
Tej Chajed. 2021. Record Updates in Coq. In *Proceedings of CoqPL 2021 (CoqPL '21).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/xxxxxxx.xxxxxxx

1 Introduction

Coq has record projections, but no way to update a field. Updating a record field is tedious and error-prone, requiring writing code that constructs a new record where all but one field comes from the old record. But records and updates to records do come up, particularly when reasoning about messy real-world objects which have lots of little bits of state that need to be managed independently.

The coq-record-update library provides a solution to this problem. First, a short example of using the library:

```
From RecordUpdate Require Import RecordUpdate.

Record X := mkX { A: nat; B: nat; C: bool; }.
Instance: Settable X := settable! mkX <A; B; C>.
Definition add3_to_B x := set B (plus 3) x.
Definition set_B_to_3 x := set B (fun _ => 3) x.
```

What's going on here? The user writes a bit of boilerplate `settable! mkX <A; B; C>` telling the library the constructor and fields (listed as projection functions) for the record. Then, `set` is usable as a function that takes a field of type `X -> T` and returns a setter function of type `(T -> T) -> (X -> X)`, which updates the field based on its current value. On top of this the library has some nice notations (for example, `set_B_to_3 x` can be written `x <|B:=3|>`) and nested updates, but these are standard notations that could be defined outside the library — the magic is really all in `set`.

The user does have to write some boilerplate in order to provide a list of field projections for `X`. This isn't too bad, but it is boilerplate we unfortunately can't eliminate (without a plugin, at least) since there's no way in Ltac or Ltac2 to go from a record type to the projection functions Coq created for that record.

The library is open-source, available on GitHub at https://github.com/tchajed/coq-record-update, and installable with opam.

Wait, how can that possibly work?! As described above this seems pretty unreasonable, from an implementation perspective. How could `set` do what it's supposed to do? The implementation uses an interesting feature of typeclasses. The Coq implementation of typeclasses is relatively simple [1]: a typeclass is basically a record whose value Coq will fill in using a proof search using only typeclass instances. These proof searches can invoke Ltac, and thus a typeclass "instance" can actually be resolved using code rather than just fixed lemmas. The result from a user's perspective is that `set x y` can actually run arbitrary Ltac code between seeing `set x` and `y`, by having a typeclass as an implicit argument to `set` and resolving that typeclass with Ltac. Of course that's exactly what we'll do.

2 Representing a generic record

Before we can see how `set` works, let's first cover the `Settable X` typeclass and how its instance is constructed. The class itself is pretty boring:

```
Class Settable R :=
  mkR: R -> R.
```

The value of `mkR` is always extensionally equivalent to the identity function. But we won't just be calling it, we'll actually use Ltac to look at the syntax of how the instance was constructed. In the example above, `mkR` is defined as:

```
mkR x = mkX (A x) (B x) (C x)
```

This is a fancy identity function that copies `x` by constructing a new record of type `X` whose fields are all drawn from `x`. I call this term the record's "eta expansion". Why is this useful? The key is that now all we need to do to update a field, say `B`, is to replace `B` in this expression with $f \circ$ `B` in order to apply an update function `f : nat -> nat`. That manipulation is purely syntactic and we can do it in Ltac.

CoqPL '21, January 19, 2021, Online

https://doi.org/10.1145/xxxxxxx.xxxxxxx

3 Building a setter using Ltac

Now we can explain what `set` is doing. First, `set` is actually just the single field of a typeclass:

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

Instances of this class are resolved by Ltac that does the following:

- Resolve `Settable R` to produce a Gallina term of type `R -> R`, the previously-described fancy identity function or eta expansion.
- Call `pattern proj` over that term to abstract over where the field is reconstructed. For our running example `X` and its field `B`, this produces a term `(fun proj => fun x => mkX (A x) (proj x) (C x)) B`.
- With Ltac pattern matching, extract the function, which is of type `(R -> T) -> (R -> R)`. This is almost what we want but `set` only passes the field value to the function, not the whole record, so for `set f` we supply $f \circ \mathrm{proj}$, which we write out in Coq as `fun r => f (proj r)`.

This Ltac runs as part of typeclass resolution by simply placing a `Hint Extern` in the `typeclass_instances` hint database. This is all that's needed to program typeclass resolution.

4 Improving the user interface

While the above implementation works, there's one big user-interface problem. What happens if the user writes `settable! mkX <B; A; C>`, which mixes up `A` and `B`? This still typechecks because the two fields have the same type, but it would now mean setting `A` sets the `B` field and vice-versa, which would be Very Bad. To address this, `Settable R` actually has an additional field *proving* that `mkR` is the identity function, and `settable!` constructs a proof (on the fly, using tactics-in-terms). As a result an incorrect declaration will fail right away. The `Setter` Ltac also has some safeguards to prevent typeclass resolution when the user tries to set a field that isn't a projection, in which case `set` would otherwise do nothing (also very bad).

5 But is this any good?

Anecdotally, it seems that SiFive is using this library (to deal with records that define the state of a RISC-V processor, particularly in the specification) as well as Bas Spitters' group in Aarhus. This makes coq-record-update my most successful Coq library (so far).

Convenience The library has some nice safeguards to prevent incorrectly listing the fields and attempting to set non-projections. However, one still has to write out the fields of the record once. We could do this automatically with a plugin, but that would make the library harder to maintain for me and harder to install for users. Ltac2 doesn't solve this either, since there's no API to query for the projection definitions.

Generality Typically I expect that users aren't concerned with generality, they just want to handle lots of fields. However, there are two interesting complications for setters: dependent fields and type parameters. In general updating dependent fields doesn't work, because we'd have to update multiple fields at the same time to preserve the type. However, updating non-dependent fields works seamlessly — the eta expansion is well-typed as written for the remaining dependent fields.

Type parameters are a little more awkward to write the `settable!` boilerplate for, but it's doable, particularly if the fields have the parameters as implicit types. Changing the type parameters in a `set` doesn't work, because it would require a more general type for `set` than '(T -> T) -> (X -> X)'.

Performance Generating setters is pretty efficient since it only involves generalizing the expression in the `Settable` instance. However, it does run once for each use of `set` in the source. If this is a problem, the user can cache the typeclass instance by declaring it ahead of time: `Instance set_B : Setter B :=_` will get resolved automatically and then get used from then on.

The generated setters are as good as manually-written ones using the record's projections. Furthemore, the term `B (set B (fun _ => 3) x)` immediately reduces to 3 (even if `x` is a variable) because the setter expression starts with `mkX` rather than pattern-matching on `x`.

Error messages `settable!` and `set` can fail for a few reasons. `settable!` fails to typecheck if fields are missing or have the wrong types; it reports a custom error if two fields of the same type are swapped. `set` generally fails because the `Setter` typeclass can't be resolved. Unfortunately in that situation we can't provide a nice error message since typeclass-resolution failure messages aren't programmable.

6 Conclusions

coq-record-update is a small library to let you update record fields. We explained how it is implemented, namely by abusing typeclasses. You can find the library at https://github.com/tchajed/coq-record-update or via the Coq opam repo. If you use it and have feedback, please let me know!

References

[1] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Theorem Proving in Higher Order Logics*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 278–293.