From Tactics to Structure Editors for Proofs

Xuanrui (Ray) Qi

Department of Computer Science
Tufts University

Off the Beaten Track '19, 19 Jan 2019

What are tactics?

Two main types of UIs for theorem proving:

- direct style: proofs are programs, so we write them just like we write programs (Agda, Idris).
- tactics: closer to how we reason about things on paper and how we think about reasoning, e.g. recursive proof term is "induction" not "recursion" (Coq, Lean, Idris *Pruviloj*).

What is my talk about?

- User interfaces for theorem provers & program editors
- How can we use *tactics* to improve theorem prover & editor UIs?

Motivation of this talk: Coq nightmare

```
Definition balanceL {nl ml d cl cr nr mr} (p : color) (l : near_tree nl ml d cl) (r : tree nr mr d cr) :
  color_ok p (fix_color l) (* important claim! *) ->
  color_ok p cr ->
  {tr : near_tree (nl + nr) (ml + mr) (inc_black d p) p | dflattenn tr = dflattenn l ++ dflatten r}.
Proof.
  destruct l as [s1 o1 s2 o2 s3 o3 d' x y z | s o d' c' cc l'].
  (* l is bad *)
  + case: p => //= cpl cpr.
    rewrite -(addnA (s1 + s2)) -(addnA (o1 + o2)).
    exists (Good Black (rnode (bnode x y) (bnode z r))).
    by rewrite /= !catA.
  (* l is good *)
  + case: p => /= cpl cpr; last by exists (Good Black (bnode l' r)).
    case Hc': c' in cpl.
    (* bad pattern (c' and p are red) *)
    - destruct l' as [|s1 o1 s2 o2 d cl' cr' c' w1 w2 l'1 l'2] => //.
      subst c'; destruct cl', cr', cr => //.
      exists (Bad l'1 l'2 r).
      by rewrite /= !catA.
    (* otherwise *)
    - subst c'; destruct cr => //.
      by exists (Good Red (rnode l' r)).
Defined.
```

Sometimes, dependently-typed programming can be quite tricky and dirty:

- byzantine invariants;
- case bloating.

Problems with each approach

- Direct style: little automation available (boring pattern-matching, boring code).
- Tactics: little idea of what you're actually doing; bad for actual programming.

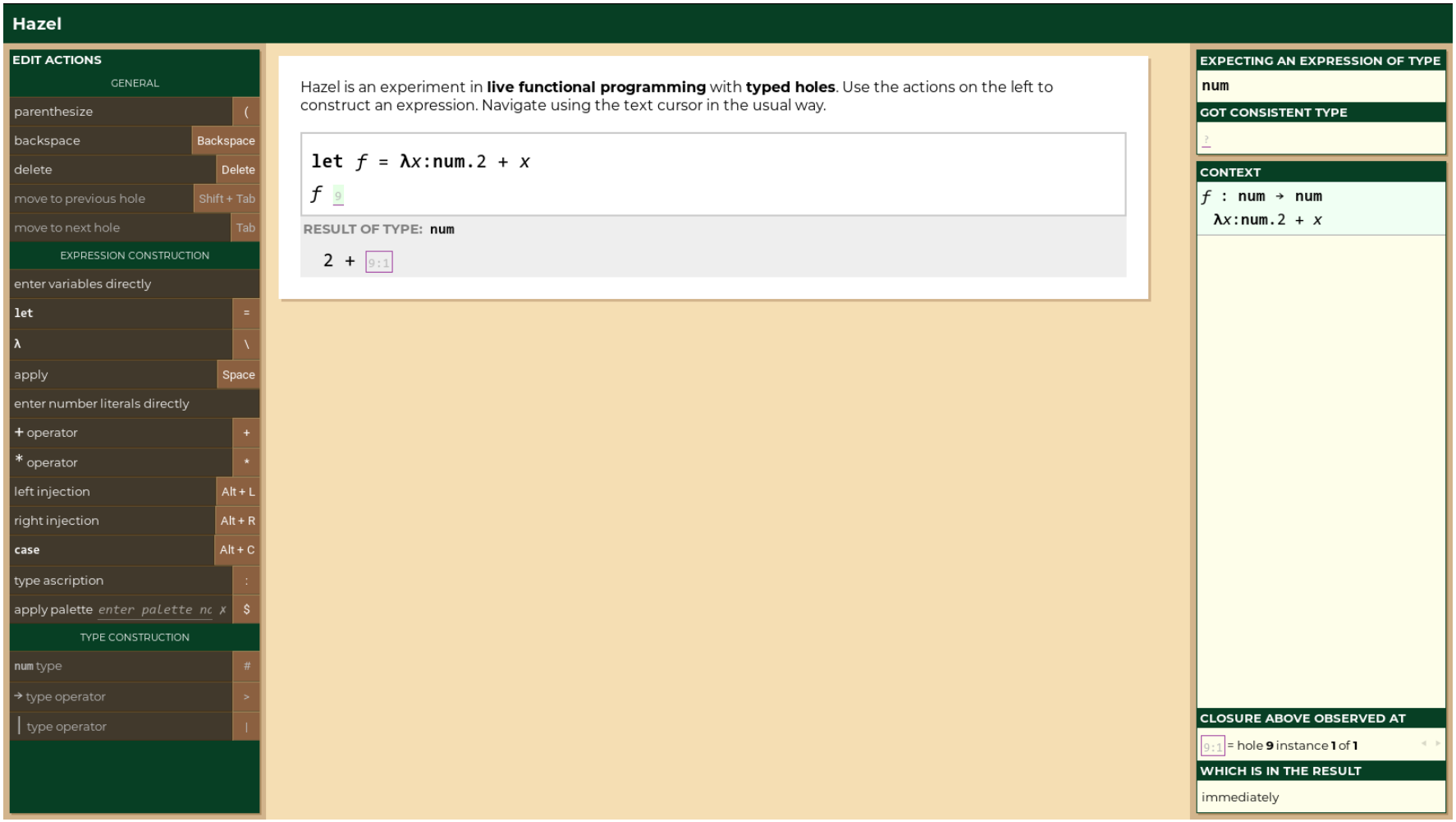
My proposal

- Design new program editors that integrate tactics as an automation method
- Design new theorem provers that allow mixing direct-style & tactics
- Study formal semantic foundations for tactics

Proposal 1: use tactics as automation

- Help automate case-splitting, especially in presence of indexed datatypes/GADTs (`induction`)
- Writing trivial clauses automatically using program search/synthesis (à la `intuition` & `crush`)
- More complex tactics might also find their uses in programming, with or without dependent types.

Holes can also help here



Proposal 2: combining tactics & programs

- Happens very often when programming with complex types: not all details we care about
- Automate chores using edit-time tactics and hide them in local, collapsable holes
- Challenge: actually generate good code
- But, do not *erase* tactics from the program: even if other parts of the program change, the housekeeping can stay the same

We may want to write programs like this...

```
Fixpoint filter {n A} (p : A -> bool) (v : vec A n) : vec A (count p v) :=
  match v with
  | VNil _ => VNil
  | VCons h t => if p h
                 then VCons h (filter p t)
                 else filter p t
  end.
```

[does not actually work in Coq]

```
filter : ∀ {n A} → (A → Bool) → Vec n A → ∃ (λ n' → Vec n' A)
filter p [] = (0 , [])
filter p (h :: t) with p h
... | true with filter p t
...                | (n'' , t') = (suc n'' , (h :: t'))
filter p (h :: t) | false = filter p t
```

But what we really want is something more like this...

```
Fixpoint filter {n A} (p : A -> bool) (v : vec A n) : vec A (count p v) :=
  match v with
  | VNil _ => VNil [ proof ]
  | VCons h t => if p h
                 then VCons h (filter p t) [ proof ]
                 else filter p t [ proof  ]
  end.
```

Proposal 3: understand the semantic foundations of tactics

- Tactics are transformations on open terms
- We should study the semantics of those transformations formally
- Benefits: create tactics that compose well, prevent tactics from breaking valid proofs/programs (like Ltac)

Building on previous work

- Hazelnut paper (POPL '17) gave semantics of some very basic "tactics" (term insertion, moving to other holes)
- Open problem: design a language to program well-behaved tactics

Putting it all together

- Traditional sorce code based editors are inadequate in a “typed” world; too much focus on minor details requireds
- Tactics can help programmers automate away those chores and focus on writing good and correct programs!
- I propose using tactic-enhanced program editors, with semantically sound tactics, to write programs.
- More automation, more editor assistance, less bookkeeping, better programs.