Lecture 31: implementing induction proofs

UNIVERSITY OF MINNESOTA
Driven to Discover®

On this course

- Project is now optional
  - To stay in the theme of the project so far:
    Sorry for not doing this sooner
  - Learning goals have been tested through midterm
- Remaining lectures are optional (I'm turning them into office-hour style lectures)
- Answers to questions I got so far:
  - If you got low grades on your project early on (say 50%) it won't hurt your standing
  - … ?

On me

- Job application got submitted on time
- Sent out 2 letters of rec this weekend (out of 3)
- Finished the project's solution write-up, gave it to TAs
- Replied to some re-sent emails
- Called my parents
- Celebrated Sinterklaas & Spouse's birthday

On today's lecture

- How to finish the project:
  - Induction
  - Match statements
  - Cleanups

Induction

- Tasks:
  - Find the type of the variable on which to do induction
  - Generate the case and induction rules
  - Call the simple prover for each step

- We do all at once in the ‘prover’ function

The ‘prover’ function

```
let rec prover types rules declarations =
      match declarations with
          | …
          | ProofDeclaration (nm, vars, Equality (lhs,rhs), Some
(Induction varnm)) :: rest
             -> (* we got an induction hint *)
                inductionproof varnm types vars rules lhs rhs
                 :: prover types ((vars,nm,lhs,rhs)::rules) rest
          | TypeDeclaration (nm, variants) :: rest
             -> (* add the type to the list of types, keep going *)
             prover ((nm, variants)::types) rules rest
          | …
```

See walkthrough for the …

How to get it to compile?

- Call prover with a types argument that's empty
- Provide a temporary definition of 'inductionproof'

Induction proof generator

- The only function left is called ‘inductionproof’.
- Note that we are starting at the final function that we want to write, and working our way toward the helper functions.
- By writing dummy functions, we can get our code to compile.
- This way of working is known a top-down approach.

- Inputs to inductionproof: basically everything.
- In practice: initially no arguments, add what you need.

Induction proof generator

```
let inductionproof varnm _types vars rules lhs rhs
=
  let typenm = "list" (* TODO *) in
  let variants = [(“Cons",["int";"list"])]
                 (* TODO *) in
  ("Proof by induction on (" ^ varnm ^ " : “
    ^ typenm ^ ") :") ::
  List.(concat (map (caseproof vars varnm typenm
rules lhs rhs) variants)) @
  ["This completes the proof by induction."]
```

Case proof

```
let caseproof vars varnm typenm rules lhs rhs
(variantnm, _) =
  let variant_namedvars = [(“?1","int");("?2","list")]
                          (* TODO *) in
  let variant_expr = mkConstructorApp (* TODO *) in
  let caserule = ([], "case", Identifier varnm,
variant_expr) in
  let ihs = [] (* TODO *) in
  let variantrules = caserule::ihs @ rules in
  ("Case "^ variantnm ^ ":") ::
  Prove.prove variantrules lhs rhs @
  ["This completes the proof of case " ^ variantnm ]
```

The conversion of types to values

- “Cons of (int * list)”
- -> “Cons”, [(“?1”, int) ; (“?2”, list)]
- -> Cons (?1, ?2)
- Intermediate step can be avoided but it’s convenient to keep for generating ihs.
- Last step is a fold-left structure
- Some of complications can be avoided by changing the ast

Rules

- Rules in this code are encoded by tuples:
  - (vars, nm, lhs, rhs)

- I’ve mixed up the order of arguments:
  - type errors are hard to read
- Consider a datatype for this:
  - type namedrule = Namedrule of (string list * …)
  - This makes your code easier to read
  - and only slightly more verbose
- Consider printing the IHs and the case rule as part of your proof (code for that is given in the walkthrough)