Repeating proof steps

Outline

- Timing of course
- Overview of functions
- How to put it all together

Course overview

- **Sunday 19th: deadline parser (print back sample.ml)**
- Monday 20: parser solution for gettingstarted.ml syntax
  - Should be useful!
- **Sunday 26:** rewriter for gettingstarted.ml deadline
- Sunday 26: *posting* full sample.ml parser solution
- Wednesday 29th: *posting* rewriter for gettingstarted.ml solution
- Friday Dec 1st: midterm
  (topics: matchings, substitutions, parsing)
- **Sunday 3rd & 10th:** deadlines for full project

Overview of functions

- attemptRewrite : string list -> equality -> expression -> expression option
  - Given a list of variables, an equality to apply, and an expression to apply it to, returns the result to applying the equality to the expression (if possible)
- tryEqualities : expression -> (string * string list * equality) list -> (string * expression) option
  - Given an expression, tries to apply each equality in the list. Returns the name of the rule that was successfully applied and the resulting expression
- performSteps : expression -> (string * string list * equality) list -> (string * expression) list
  - Applies tryEqualities on the new expression until it returns None, gathers the results as a list
- produceProof : equality -> (string * string list * equality) list -> string list
  - Attempts to prove the equality by breaking up the lhs and the rhs and calling performSteps on each side. Inserts ??? if the lhs and rhs end up staying different
- produce_output_simple : declaration list -> string

attemptRewrite : string list -> equality -> expression -> expression option

- Use the matching function to match the lhs of the equality to the expression
- If there is no match, use recursion:

```
match match_expressions variables lhs expr with
| … (* matches! *) -> Some (…)
| … (* not a match *) -> (match expr with
   | App (fn,arg) -> (match attemptRewrite … arg with
      | Some v -> Some (App (fn, v))
      …
```

- In my datatype, I found it convenient to use mutual recursion, syntax:
  - let rec fn1 …
    and fn2 …
  - The use of ‘and’ instead of ‘let rec’ allows for fn1 to call fn2

tryEqualities : expression -> (string * string list * equality) list -> (string * expression) option

- tryEqualities expr equations
  returns the pair (eqn, expr’)
  where eqn is the name of the equation that applied
  and expr’ is the rewritten expression
- Simply calls the attemptRewrite function for each of the equalities

performSteps : expression -> (string * string list * equality) list -> (string * expression) list

(name, list of its variables, equality)
Consider making another type for this!

- performSteps expr eqns
  returns a ‘one-sided rewrite proof’, for doing this proof, for example:
- map f (map g lst)
  = {C}
  map f (map g (Cons (x, xs))
  = {A}
  map f (Cons (g x, map g xs))
  = {A}
  Cons (f (g x), map f (map g xs))
  = {D}
  Cons (f (g x), map (compose f g) xs)
- performSteps (parse “map f (map g lst)”) eqns would return :
- [(“C”, parse “map f (map g (Cons (x, xs))”) ;
  (“A”, parse “map f (Cons (g x, map g xs))”) ; .. and so on

produceProof : equality -> (string * string list * equality) list -> string list

- produceProof eq knownEqs
  produces a proof that shows that eq is true
- Suggestion: define a helper function, call it with the output of ‘performSteps lhs’ and ‘performSteps rhs’
- The lists ‘performSteps lhs’ and ‘performSteps rhs’ should end on the same expression for the proof to be valid.
  - If they don’t, it’s useful to have a step called “???”
  - The last *n* steps in both lists might be identical, if so, those steps can be removed!
- I produce a list with a string for each line,
  then String.concat “\n” them later.
  Hence the “string list” result type
  (choosing “string” is fine too)

produce_output_simple : declaration list -> string

- Traverse over the file (which is a declaration list)
- Keep an accumulator (helper function!) with the equalities gathered so far
- If a proof statement with axiom-hint is encountered, add it to the equalities gathered so far
- If a proof statement without hint is encountered, prove it first and then add it to the equalities gathered
- Put newlines in the end-results

Putting it all together: adding a switch to bin/main.ml

- In the main.ml function, we have code that calls ‘print_all’. Here is the call tree:
  - printback_file calls print_all
  - printfile calls printback_file (wrapped inside a protectx)
  - speclist refers to ‘printfile’, adding it as one of the arguments we can call.
- We could duplicate all of the above code, or ….

Putting it all together: adding a switch to bin/main.ml

- In the main.ml function, we have code that calls ‘print_all’. Here is what we can change:
  - printback_file calls print_all « give it an argument ‘fn’
  - printfile calls printback_file « pass the ‘fn’ argument (wrapped inside a protectx)
  - speclist refers to ‘printfile’
    « change to ‘(printfile print_all)
- While we’re at it, perhaps change the names of these three functions into something more meaningful

Note on data-types

- Data-types are everything!
  - They drive your recursive structure
  - Simpler data-types means simpler functions
  - The perfect data types matches your recursive calls one-to-one
- Don’t be afraid to:
  - add a ‘duplicate’ type and use conversion functions
  - refactor old types into new ones

Note on debugging

- Save what you test
  - For your parser, you may already be getting a fine selection of files with expressions/statements
  - Calling the matching, substitution, etc..
    - Create values in ways that makes it reusable:
    - At some point you will leave utop!
    - Make sure you can rerun your test when you reopen utop