Lecture 29:
Some unit tests for all your functions

Sebastiaan Joosten

UNIVERSITY OF MINNESOTA
Driven to Discover®

This lecture

- How to set up tests if I don’t know your ast.ml ?
- We go over every function you need to write…
  - substitution
  - matching
  - doing a proof-step
  - doing many proof-steps
  - present the proof
- Adding a --simple switch to bin/main.ml

Caveat

- Make sure your code compiles after each step!
- If your code doesn’t compile, we cannot use utop, or test in any other way!

Setting up test-cases: The problem

- Let’s say we want to test whether our ‘matching’ function returns a substitution successfully:
  - assert (match_expression variables pattern goal = Some substitution)
  - let’s say we want to see if the pattern ‘foo x y’ matches ‘foo (bar z) (bar z)’ with variables x and y.
  - variables = [“x”; “y”]
  - pattern = ???
  - goal = ???
    - these depend on your data-type!

Setting up test-cases: the solution

- You've written a parser, so let's use it!

```
let parse_expression (s : string) : expression =
  let lexbuf = Lexing.from_string s in
  let ast = Parser.expression_eof
              Lexer.token lexbuf in ast
```

- Add a parse-rule for **expression_eof** in parser.mly:

```
expression_eof:
| e = expression ; EOF {e}
```

- Tell menhir to generate a function for it by adding this on the top of parser.mly:

```
%start expression_eof
%type <expression> expression_eof
```

- We can also create a 'parse_equality' for equalities and so on (I'll assume you'll figure out how to do that)

Back to matching…

- assert (match_expression variables pattern goal
                                        = Some substitution)

- match_expression [“x”; “y”]
     (parse_expression “foo x y”)
     (parse_expression “foo (bar z) (bar z)”)
  = Some …?
- the … depends on how you’ve implemented things!
  - This time it’s your substitution module!

Back to matching…

- assert (match_expression variables pattern goal
                                        = Some substitution)

- match_expression [“x”; “y”]
      (parse_expression “foo x y”)
      (parse_expression “foo (bar z) (bar z)”)
  = Some (Substitution.merge
                      (Substitution.singleton “x”
                                  (parse_expression “bar z”))
                      (Substitution.singleton “y”
                                  (parse_expression “bar z”)))

This is messy!

Some code for your substitution module

```
let print_subst (s : t)
   =  MM.iter (fun k v -> print_endline
   (k ^ " -> " ^ string_of_expression v)) s
```

- Or if you're using lists of key-value pairs, you can use:

```
let print_subst (s : t)
   =  List.iter (fun (k, v) -> print_endline
   (k ^ " -> " ^ string_of_expression v)) s
```

Back to matching…

- match_expression [“x”; “y”]
  (parse_expression “foo x y”)
  (parse_expression “foo (bar z) (bar z)”)
  = Some <abstr>.t
  x -> bar z
  y -> bar z

- It’ll be up to you to call and interpret your printing function:
  - let Some result = match_expression …;;
    Substitution.print_subst result
- This might print things in a different order, print expressions differently, etc.

Back to matching…

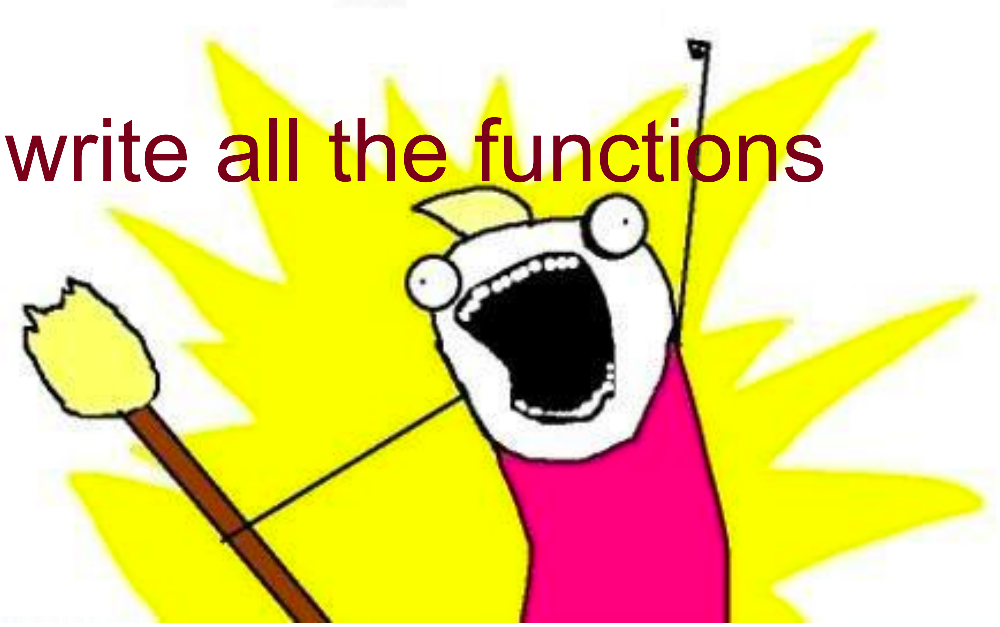
- match_expression [“x”; “y”]
  (parse_expression “foo x y”)
  (parse_expression “foo (bar z) (bar z)”)
  = Some <abstr>.t
  x -> bar z
  y -> bar z

- Ill write parse whenever I mean parse_expression

Back to matching…

- match_expression [“x”; “y”]
    (parse “foo x y”)
    (parse “foo (bar z) (bar z)”)
  = Some <abstr>.t
  x -> bar z
  y -> bar z

- Ill write parse whenever I mean parse_expression write all the functions



- We go over every function you need to write…
  - **substitution**
  - matching
  - doing a proof-step
  - doing many proof-steps
  - present the proof

Substitution

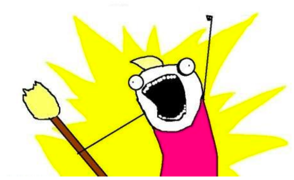
- This is not a function, it's actually multiple:

- empty
- singleton
- merge
- find

- Example test:
  merge empty empty = Some empty
- Again: maybe your types are slightly different, and you'll need to adjust accordingly. Check that you understand what we're trying to test! For instance, you might want to type this if you put everything in a module:
- Substitution.(merge empty empty)
- More ideas?

Things to test for your substitution module

- merge (singleton “k” (parse “foo”))
  (singleton “k” (parse “foo”))
  = Some (singleton “k” (parse “foo”))
- merge (singleton “k” (parse “foo”))
  empty = Some (singleton “k” (parse “foo”))
- merge (singleton “k” (parse “foo”))
  (singleton “k” (parse “bar”)) = None
- find “k” (singleton “k” (parse “foo”)) = parse “foo”
- let Some res
  = merge (singleton “k” (parse “foo”)) (singleton “l” (parse “bar”))
  find “k” res = parse “foo”
  find “l” res = parse “bar”



- We go over every function you need to write…
  - substitution
  - **matching**
  - doing a proof-step
  - doing many proof-steps
  - present the proof

Implementing matching:

- matching variables pattern goal:
  - match pattern with
    - variables: singleton
    - constants: must match goal exactly
    - applications: goal must be an application too
      - recurse on function + arguments
      - merge the substitutions

Testing matching

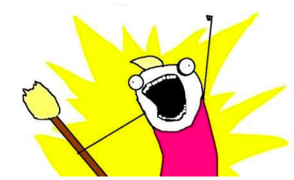
- match_expression [“x”; “y”]
  (parse_expression “foo x y”)
  (parse_expression “foo (bar z) (bar z)”)
  = Some <abstr>.t
  x -> bar z
  y -> bar z

- What else should we test?

Testing matching

- match_expression [“x”; “y”]
    (parse_expression “foo x x”)
    (parse_expression “foo (bar z) (bar y)”)
  = None

- match_expression [“x”; “y”]
    (parse_expression “foo x (foo y)”)
    (parse_expression “foo (bar z) y”)
  = None



- We go over every function you need to write…
  - substitution
  - matching
  - **doing a proof-step**
  - doing many proof-steps
  - present the proof

Doing a proof step

- Doing a proof step based on a single equality:
- attempt_rewrite variables eq expr
- variables: list of variables
- eq: Input equality, (lhs, rhs)
- expr: expression we’re trying to rewrite

- Call the matching function and see if expr matches the pattern lhs
- If not, try it for subterms of expr
- If successful, we’ll need to **apply the substitution to the rhs**!

Doing a proof step: substitution function

- substitute variables t expr = expr’

- substitute [“x”]
      (singleton “x” (parse “foo x”))
      (parse “bar x y x”)
  = parse “bar (foo x) y (foo x)”

- writing the function: recursion (match) on expr

Doing a proof step: single step

- tryEqualities equalities expr = Some (nm, expr’)
- … I’ll define a list of equalities to use for testing. Each equality will have a name, a list of variables, a left-hand-side expression and a right-hand-side expression:
- let myEqualities =
   [(“eq1”, [“x”], parse “foo x x”, parse “bar x”)
   ; (“eq2”, [“x”;”y”], parse “bar x y”, parse “y”)
   ; (“eq3”, [“x”], parse “bozo x (foo x)”, parse “foo (bar x x)”]

Doing a proof step: single step

- tryEqualities myEqualities (parse “foo x x”)
  = Some (“eq1”, parse “bar x”)
- tryEqualities myEqualities (parse “foo (bar a b) c”)
  = Some (“eq2”, parse “foo b c”)
- tryEqualities myEqualities
            (parse “bozo (bar a) (foo (bar a)”)
  = Some (“eq3”, parse “foo (bar (bar a) (bar a))”)

- let myEqualities =
   [(“eq1”, [“x”], parse “foo x x”, parse “bar x”)
   ; (“eq2”, [“x”;”y”], parse “bar x y”, parse “y”)
   ; (“eq3”, [“x”], parse “bozo x (foo x)”, parse “foo (bar x x)”]

- We go over every function you need to write…
  - substitution
  - matching
  - doing a proof-step
  - **doing many proof-steps**
  - present the proof

Doing a proof step: multiple step

- performSteps equalities expr = .. list of steps

- Writing the function:
- match on the output of tryEqualities:
    - if None, return the empty list
    - if Some (nm,e), return it at the head of a list that you continue recursively

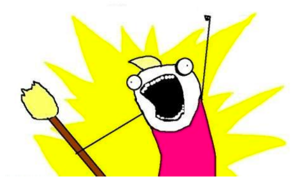
Doing a proof step: multiple step

- performSteps equalities expr = .. list of steps
- performSteps myEqualities

```
          (parse “bozo a (bar (foo b) (foo a)) a”)
= [ (“eq2”, parse “bozo a (foo a) a”)
  ; (“eq3”, parse “foo (bar a a) a”)
  ; (“eq2”, parse “foo a a”)
  ; (“eq1”, parse “bar a”) ]
```

- let myEqualities =

```
 [ (“eq1”, [“x”], parse “foo x x”, parse “bar x”)
 ; (“eq2”, [“x”;”y”], parse “bar x y”, parse “y”)
 ; (“eq3”, [“x”], parse “bozo x (foo x)”, parse “foo (bar x x)” ]
```



- We go over every function you need to write…
  - substitution
  - matching
  - doing a proof-step
  - doing many proof-steps
  - **present the proof**

Present the proof

- prover_simple equalities equality =
  [list of strings that outputs the proof]

- writing the function:
  - use a helper function, call it like this:
    ```
    produceProof
        (performSteps equalities lhs)
        (performSteps equalities rhs) lhs rhs
    ```
  - … start by writing a function that doesn’t remove duplicate proof-parts, fix that later

Present the proof

- prover_simple equalities equality =
  [list of strings that outputs the proof]

- tests:
  - We’re almost at the point where we can call our executable!

This lecture

- How to set up tests if I don’t know your ast.ml ?
- We go over every function you need to write…
  - substitution
  - matching
  - doing a proof-step
  - doing many proof-steps
  - present the proof
- **Adding a --simple switch to bin/main.ml**

Main.ml

- I secretly (read: accidentally) gave two versions:
    - on canvas, there's the posted main.ml
        - has 'print_all' baked in
        - instructions on how to generalize were already given
    - in the speclist, there's this line for the arguments:

```
let speclist =
  [( "--printback"
   , Arg.String print_file
   , "Print the parsed file back out")]
```

Main.ml in last week's solutions

- In last week's solutions there's the generalized main.ml
  - has 'print_all' mentioned explicitly
- in the speclist, there's this line for the arguments:

```
let speclist =
  [( "--printback"
   , Arg.String (with_file print_all)
   , "Print the parsed file back out")]
```

- We can simply add another line to it…

Main.ml with an extra switch

```
let speclist =
  [("--printback"
   , Arg.String (with_file print_all)
   , "Print the parsed file back out")
  ;("--simple"
   , Arg.String (with_file (fun x -> ()))
   , "Parse the file, but don't print anything")]
```

- Note that this is not the right function to plug in (though it matches the description).

Type of the function …

- Gets in a list of declarations
- Returns ()

- side effect: print something

- in our case: print a list of proofs

- implementing it goes something like this:

```
let produce_output_simple (lst : declaration list)
  = print_endline (String.concat “\n\n"
      (List.map (String.concat “\n")
                (proofs_of_simple [] lst)))
```

Recursive helper

Proofs of (simple)

- Get a list of declarations
- Keep a running list of equalities known so far, initially empty (either by proving them or assuming them)

```
let rec proofs_of_simple eqs (lst : declaration list) =
match lst with
  | [] -> []
  | (ProofDeclaration (nm,vars,eq,hint))::decls
     -> (match hint with
          | None -> (("Proof of "^nm^": ") :: ["TODO"])
                :: (proofs_of_simple ((nm,vars,eq)::eqs)
                                 decls)
          | _ -> proofs_of_simple ((nm,vars,eq)::eqs)
                               decls )
  | _::decls -> proofs_of_simple eqs decls
```

Proofs of (simple)

- Get a list of declarations
- Keep a running list of equalities known so far, initially empty (either by proving them or assuming them)

```
let rec proofs_of_simple eqs (lst : declaration list) =
match lst with
  | [] -> []
  | (ProofDeclaration (nm,vars,eq,hint))::decls
    -> (match hint with
         | None -> (("Proof of "^nm^": ") :: ["TODO"])
              :: (proofs_of_simple ((nm,vars,eq)::eqs)
                                    decls)
         | _ -> proofs_of_simple ((nm,vars,eq)::eqs)
                                 decls )
  | _::decls -> proofs_of_simple eqs decls
```

Likely type error: list of variables needs to be converted

Summary

- How to set up tests if I don’t know your ast.ml ?
- We go over every function you need to write…
  - substitution
  - matching
  - doing a proof-step
  - doing many proof-steps
  - present the proof
- Adding a --simple switch to bin/main.ml