

Solutions to Homework 3

Chris Fensch & Chris Stork

September 2, 2004

These answers are a bit shorter than for previous homeworks.

1 Matching, Equality, and ML

What we called multi-match, Mitchell calls non-linear match. In exercise 5.3 he discusses why ML does not use non-linear matching. Note that this is all about the fact that, even if x and y have the same type, you cannot always perform the comparison $x = y$. (20 Points)

- (a) Almost the same as his (a), but replace the $+$ with $*$.
- (b) What he's really asking for in (b) is "What additional language facility or functionality do you need to use to implement the `eq` function?". This is not deep. It's the obvious answer!
- (c) For part (c) what he's going for is: "The ML language designers decided to leave non-linear matching out of the language. Is this a serious restriction or can a programmer easily program around that?" Remember that you can refer to previous lectures to answer part of this question.
- (d) First say why it's impossible for ML to decide in general whether two functions are equal. Then, a one sentence bla answer to Mitchell's question is fine here.
- (e) Additional part: Why didn't we have these kind of problems when writing `mmatch`? How was our notion of equality different then in the previous part of this question? In which line of `query.scm` does the relevant equality test take place? (Look twice!)

Answer:

```
(a) fun f (pair : int * int) =
      if #2 pair = 0
      then #1 pair
      else if #1 pair = 0
           then #2 pair
           else #1 pair * #2 pair;
```

- (b) Well, in a way combining destructors and if-then-else does work:

```
fun eq (pair : int * int) =
  if #1 pair = #2 pair
  then true
  else false;
```

Hmm, but the `int` type is too specific. So, we try

```
fun eq (pair : 'a * 'a) =
  if #1 pair = #2 pair
  then true
  else false;
```

and get the error message

```
! Toplevel input:
!   if #1 pair = #2 pair
!           ^^^^
! Type clash: expression of type
!   'a * 'a
! cannot have type
!   {1 : 'b, ...}
```

Weird, what's this `'b`? It's an *equality type*, i.e. a type on which equality is defined. So, let's try and use that:

```
fun eq (pair : ''a * ''a) =
  if #1 pair = #2 pair
  then true
  else false;
```

Great that works. So, the “obvious” answer would be: In order to provide non-linear pattern matching the language would need to provide an equality operation between arbitrary types.

- (c) We showed above that the programmer can program around the restriction of linear patterns by using if-then-else and explicit equality checks. The general solution works just like our `mmatch` function.
- (d) It's impossible to compare arbitrary functions (unless it's just by name/location, which is relatively arbitrary, even though Scheme does it this way) since that implies comparing them on all possible input values. But even comparing two functions on one input value is not computable according to the Halting Problem. The designers of ML thought that they can't solve the problem of non-linear patterns in general in a satisfactory way and they know that it's easy to program around it (see above). That's why they dodged.

- (e) `mmatch` simply used Scheme's `equal?` and since we operate on symbols or literals our notion of equality was *equality by name*. The relevant equality check takes place in line 120: that's where a previous binding to a logical variable is compared against a new binding.

Some problems with equality are listed below. It was not necessary to go into these in depth in your homework solution. Some of these involves ideas that are not covered in lecture or the reader.

- Testing two tuples, lists or records to see if they are equal requires computational effort to test their components for equality. For example, evaluating the expression `(a, (b, (c, d))) = (a, (b, (c, d)))` involves testing `a = a`, `b = b`, `c = c` and `d = d`. Similarly, according to the ML definition of equality on lists, testing whether two lists are equal involves traversing both lists to see if they are the same length and, if so, if corresponding elements are equal.
- Some types do not admit equality. Functions, for example, are never tested for equality in ML. The rationale for this design decision is that this avoids arbitrary decisions on what functional equality should mean. True function equality is not computable; you would have to try all arguments, and even then you couldn't be sure because you wouldn't know how long to compute before deciding one of the functions was not defined on some argument. User-defined abstract types also do not admit equality unless the implementor defines an equality function for it.
- References admit equality on a pointer basis, much like Lisp's `eq?`. More specifically, if we ask whether two pointers are equal, the test is whether they point to the same memory location (or "reference cell," in ML terminology), not whether the locations they point to contain equal values. As a result, the value of the expression `ref 1 = ref 1` is `false`, since evaluating each subexpression `ref 1` generates a different reference cell containing the number 1. This eliminates problems in testing equality for circular lists, which some of you were concerned about.
- Since some types admit equality and others do not, equality has a more complicated typing property than anything else in ML. Although we did not discuss this, ML uses a different kind of type variable for so-called "equality types". A function such as `fun f(x,y) = x=y`; has type `a bool`, meaning that `f` may be applied to any pair of arguments belonging to the same equality type. For example, we can apply `f` to two pairs of integers, `(x, y)` and `(u, v)`, with result `true` iff `x = u` and `y = v`. However, `f(f, f)` is considered a type error, since `f` is a function and function types are not equality types. A reasonable treatment of non-linear patterns would therefore involve equality types.

2 Unification

You've seen two presentations of how to perform unification. First, the more abstract unification algorithm according to Martelli-Montanari and, second, the concrete im-

plementation as `unify` procedure in `infer.scm`. This exercise is meant to confirm that they lead to the same result.

Consider the following equation for unification.

$$f(a, x, g(z, b), z) = f(x, x, y, g(x, h(c, c)))$$

- (a) Derive the MGU by following the steps of Martelli-Montanari's algorithm. (10 Points)
- (b) Transcribe the equation so that `unify` can present you with the result. Confirm its correctness by comparing it to your manual calculations above. Please include a copy of invocation and result to demonstrate your success. (10 Points)

Answer:

(a)

$f(a, x, g(z, b), z) = f(x, x, y, g(x, h(c, c)))$	Apply rule 1.
$a = x; x = x; g(z, b) = y; z = g(x, h(c, c))$	Apply rule 4 to $a = x$.
$x = a; x = x; g(z, b) = y; z = g(x, h(c, c))$	Apply rule 3 to $x = x$.
$x = a; g(z, b) = y; z = g(x, h(c, c))$	Apply rule 4 to $g(z, b) = y$.
$x = a; y = g(z, b); z = g(x, h(c, c))$	Apply rule 5 with $[x/a]$.
$x = a; y = g(z, b); z = g(a, h(c, c))$	Apply rule 5 with $[z/g(a, h(c, c))]$.
$x = a; y = g(g(a, h(c, c)), b); z = g(a, h(c, c))$	

(b) > (unify
 '(f a ?x (g ?z b) ?z)
 '(f ?x ?x ?y (g ?x (h c c))))
 (((?x . a)
 (?y g (g a (h c c)) b)
 (?z g a (h c c))))

3 Family Relations among the Simpsons

Add some rules to the `simpsons-facts` for `grandparent`, `granddad`, `sister`, and `sister-in-law`. Start with the template file `infer-relations-template.scm`. It contains some extra facts (sibling and married) in the DB and it already has some test cases plus examples of the desired query results. (Your result order might vary though.)

Hand in your modified file as `infer-relatives.scm`. (Draw—for yourself—the search trees corresponding to your new queries!) (10 Points)

Hint: If it happens to you that each daughter is a sister of itself that is fine, but if you want to get extra credit look at question 1, which hints at a way to define equality with a rule. For this, also remember what I said about a rule body that only contains '(and)'!

Answer:

The following rules should be added to the DB.

```
(← (equal ?x ?x)
  (and))
(← (father ?x ?y)
  (and (parent ?x ?y) (male ?x)))
(← (mother ?x ?y)
  (and (parent ?x ?y) (female ?x)))
(← (sibling ?x ?y)
  (and (father ?f ?x) (father ?f ?y)
        (mother ?m ?x) (mother ?m ?y)
        (not (equal ?x ?y))))
(← (sister ?x ?y)
  (and (sibling ?x ?y)
        (female ?x)))
(← (grandparent ?gp ?gc)
  (and (parent ?gp ?p)
        (parent ?p ?gc)
        ))
(← (granddad ?gd ?gc)
  (and (grandparent ?gd ?gc)
        (male ?gd)))
(← (sister-in-law ?s ?x)
  (and (sister ?s ?y)
        (married ?y ?x)))
```

4 Queries & Backtracking

Understand the append-to rules at the end of infer.scm! Be sure to understand the meaning of the improper list notation. (10 Points)

- (a) *Define rules for member relation as described on page 487 in Mitchell, add them to the bottom of infer.scm, use tests to assert their correctness, and hand the resulting file in as infer-member.scm!*
- (b) *What happens if you reverse the order of the two rules? Why?*

Again make sure for yourself that you understand how the backtracking works.

Answer:

```
(a) (define member-rules
      '((← (member ?x (?x . ?ignore))
          (and))
        (← (member ?x (?ignore . ?tail))
```

```

        (member ?x ?tail))
    ))

(set! *db* member-rules)

(test member
  ((query-infer '(member a ()))
   ())
  ((query-infer '(member a (b c d)))
   ())
  ((query-infer '(member a (a b c)))
   (()))
  ((query-infer '(member a (c b a)))
   (()))
  ((query-infer '(member ?x (a b c)))
   (((?x . a)) ((?x . b)) ((?x . c))))
  )

;; rev-member
(define rev-member-rules
  '(((<- (member ?x (?x . ?ignore))
        (and)
        (<- (member ?x (?ignore . ?tail))
            (member ?x ?tail))
        )))

(set! *db* member-rules)

(test rev-member
  ((query-infer '(rev-member ?x (a b c)))
   (((?x . c)) ((?x . b)) ((?x . a))))
  )

```

- (b) As seen by the test cases, the order of the results in a give-me-the-members-of-that-list type of question is being reversed. This happens because back-tracking now applies the rule “go deeper into the list” first, once it has reached the end it will then apply the rule “check for a match”.

5 Getting started with ML

Write different versions of our *app* function (appending two lists) in ML. (10 Points)

- (a) the old-fashioned (Scheme-kinda) way
- (b) the ML way with pattern matching
- (c) the iterative way (no recursion allowed)

Which one feels best to you? ;-)

Answer:

- (a) `fun app [] l = l
| app (h::t) l = h::(app t l);`
- (b) `fun app x y = if x = [] then y
else (hd x)::(app (tl x) y);`
- (c) There's two ways to implement `app x y`:
- i. Reverse `x` and then append with a while loop.
 - ii. Construct an iterative `nth` function and use it. That's done below.

```
fun app x y =
  let
    val i = ref (length x)
    val result = ref y
    fun nth i l =
      let val ir = ref i
          val lr = ref l
        in
          while not (!ir = 1)
          do ( lr := tl (!lr);
              ir := !ir - 1
            );
          hd (!lr)
        end;
  in
    while not (!i = 0)
    do ( result := nth (!i) x :: (!result);
        i := !i - 1
      );
    !result
  end;
```

6 Map on Trees

This is the same as Mitchell's question 5.4. Hand in the code as `maptree.ml`. Your program should end with the statement *(20 Points)*

```
printVal (maptree f (NODE(NODE(LEAF 1,LEAF 2),LEAF 3)));
```

Answer:

```
(a) datatype 'a tree = LEAF of 'a
    | NODE of ('a tree * 'a tree);

fun maptree f (LEAF(s)) = LEAF(f s)
  | maptree f (NODE(x,y)) = NODE (maptree f x, maptree f y);

printVal (maptree f (NODE(NODE(LEAF 1,LEAF 2),LEAF 3)));
```

- (b) ('a -> 'b) -> 'a tree -> 'b tree. Since no assumptions are being made about function f , except that it takes one argument and returns another one, it can return a different type. For example, f could be a function that converts an integer value to a real value. Hence the function returned by `maptree` will convert an integer tree into a real tree.

7 Types & Garbage Collection

Same as Mitchell's 6.3.

(10 Points)

Answer:

Adding type casting to a garbage collected language makes garbage collection more difficult (though not necessarily impossible). Several issues that complicate garbage collection:

- (a) For some garbage collection systems, we may need to know the size of a memory block. If we use the type of the variable to know the size of the block, then a cast may make the garbage collector think the block has the wrong size. The confused garbage collector may reclaim the wrong amount of memory. (The solution is to store the size in the header of the memory block.)
- (b) There is also a problem when converting pointers to non-pointers, and non-pointers to pointers.
 - If we convert a pointer into an integer, then the block referenced by the pointer may now be garbage. The garbage collector may not collect the garbage. This by itself is not a major problem, however. Garbage collectors are already conservative, i.e. not all garbage is collected; casting just adds to the set of uncollected garbage.
 - If we convert an integer into a pointer, then the collector may think the pointer points to non-garbage when it does not, so it may scan that (invalid) memory block to find pointers to other memory blocks. Those memory blocks may be garbage, but the collector will not be able to collect them.

Neither type of cast is a major problem, but when the two are combined, then we can have a serious error. To demonstrate the problem with casting and garbage

Solutions to Homework 3

collection, we must *both* cast a pointer to an integer, and cast an integer to a pointer:

```
char* s = new char[50]; // allocate block
int i = (int)s + 18;    // cast to an integer, and disguise it
s = NULL;              // remove existing pointer to s

// run garbage collector here; it sees no pointers to the block,
// so it frees that block

s = (char*)(i-18);     // cast the integer back into a pointer

// at this point, we can access the block
```

By our definition of garbage, the block is *not* garbage because it will be accessed again. However, the garbage collector has reclaimed the memory.

In general, when non-pointer types can store pointer values, and vice-versa, it becomes difficult to tell what is garbage and what is not. In order to avoid collecting live memory (and thus crashing the program) its important to modify the memory allocator to store extra information with variables and allocated memory so that, at run time, the garbage collector can determine which values are memory addresses and how many bytes to deallocate when memory becomes garbage. One point was given for the conclusion that GC would become hard or complicated with type casting. One point was given for recognizing that the problem was caused by casts to or from pointer types. One point was given for identifying one of the three problems listed above and two points were given for a clear discussion of what the problem was.