Solutions to Homework 2

Chris Fensch & Chris Stork

August 28, 2004

1 equal? vs eqv? vs eq?

For each of the appearances of `equal?` *in the following lines of the file* `query.scm` *state and justify if* `equal?` *could be replaced by* `eqv?` *or* `eq?`.

Answer:

l. 33 If you restrict the relational key-words in the data base to only symbols (as it is the case with our DB) then all three forms (`equal?`, `eqv?`, and `eq?`) will produce equivalent results. As soon as you allow the relations to be numbers or lists only `equal?` will work.

l. 83 `equal?` could not be replaced by either of the other options in this case because you can be dealing with lists. For example, if you have `(match '(male homer) '(male homer))` you should get an exact match. However, the test on line 83 will only be true in this case if you use `equal?`.

l. 115 `equal?` again could not be replaced by either of the other options at this point in the code. This line does the exact same thing in `mmatch-rec` as line 83 does in `match`, so again, because input lists that have equal contents but not the same physical address `eqv?` and `eq?` will not return true when they should.

l. 120 The left-hand expression will always be a symbol, so only `eq?` is needed. Even if the right-hand expression is a list, a symbol is not equal to a list, so `eq?` gives the expected answer.

l. 237 Only `equal?` will work as desired in this case, because the two matching algorithms return lists, which may be identical, but have different addresses, so `eqv?` and `eq?` will not find them to be equivalent.

(30 Points)

2 No global DB

Your instructor was too lazy to write the embedded DB implementation in a purely functional style. Please do it for him! More precisely, hand in a Scheme program that does not contain the global variables `*db*` and `*fresh-vars-counter*` and that implements `query-infer` and everything it depends on in a purely functional style.

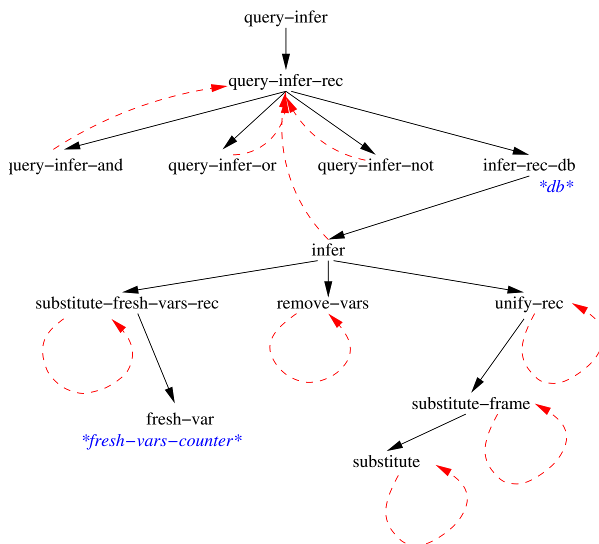
Try to remove `*db*` first! It's much harder to remove the usage of `*fresh-vars-counter*`. You do not have to remove the latter in order to get full credit for this question. But you may chose to do so for extra credit.

To test your code, adapt and include the Simpsons tests and the ad hoc tests for `substitute-fresh-vars-rec`. Please hand in the code, including running test cases (that means you should include a copy of the `test` macro), as a file `pure-infer.scm`.

Also write a few sentences explaining in which respect this code is better or worse than the original. Can you think of any other reason than lazyness why the code was presented in the original form?

Extra info from email: *You are already eligible for extra points if you just describe how to remove the global variable `*fresh-vars-counter*`. It is especially important though, that you describe what causes the additional problems compared to removing `*db*`. It would be even better if you offer example code showing how to overcome these problems.*

Answer:



The solution to removing `*db*` was to simply add a DB argument to every function that might potentially need access to the DB. This includes functions that might call other functions that might need access and so on. In the nice *call graph* above you can see that only `infer-rec-db` accesses `*db*` directly. Functions needing an

extra DB argument can now be determined by following the call arrows in the reverse direction. (Technically, we are determining the *transitive closure* of `infer-rec-db` callers.) This means that all functions above and including `infer` need a new DB argument. An alternative way of solving this problem would be to simply remove the definition of `*db*`, run the program, and deal with each complaint of the Scheme interpreter by adding the new DB argument or adapting the tests! (30 Points)

The additional problem with removing `*fresh-vars-counter*` is that, in contrast to the `*db*` a counter is changing its value. Therefore, if `*fresh-vars-counter*` is incremented in `fresh-var` (see call graph), its changed value needs to be communicated back to `substitute-fresh-vars-rec` and indeed again to the whole transitive closure of its callers, which additionally includes everything above of and including `infer`. The solution is to modify all affected functions to return a data structure which contains the regular result and the current counter. In Scheme's case this data structure is of course a pair. This sounds easy but when you implement it, it becomes painful[1] due to tedious packing and unpacking. One specific problem is the use of our `reap` function with the new functions. Look at `fvc-reap` for one possible solution.

See the two files `pure-infer-no-counter.scm` and `pure-infer-no-global-db.scm` for implementations of both modifications. We suggest that you compare[2] these files to the original `pure-infer.scm` in order to see what was changed. BTW, `pure-infer.scm` has been updated to fix one minor bug and some comments.

Pros and cons of the purely functional code: (10 Points)

- Explicit dependencies in form of arguments and return values
- OTOH, this might make argument lists too long and if the argument is supposed to be constant during all of the exectution (as in `*db*`'s case) it might just be boilerplate.
- No side-effects. For example, in question 3 the additional `let` (to make `t` evaluate only once) would not be necessary if the test cases were purely functional to begin with.
- OTOH, handing around state as with `*fresh-vars-counter*` can be tedious.
- All in all, picking the right balance between functional and imperative features seems the right approach to make code more understandable.

3 Macros

Consider the test macro used in the files `query.scm` and `infer.scm`:

[1] At least given the presented subset of Scheme it's painful. A real Scheme hacker would use features like `values` (see section 6.4 of R5RS) and maybe some macros to perform the modifications on a higher level.

[2] The unix `diff` command and any of its GUI versions are very well-suited for this task.

```
1 ;; This test macro takes a test unit name u and a list of (test result) lists.
2 ;; If a test does not produce the expected result it displays a failure report.
3 (define-syntax test
4   (syntax-rules ()
5     ((test u (t r)) ;; test unit u with test t and expected result r
6      (let ((tr t)) ;; compute test results tr
7        (if (not (equal? tr 'r))
8            (begin
9              (display "\nTEST ")
10             (display 'u)
11             (newline)
12             (display 't)
13             (display "\nEXPECTED:    ")
14             (display 'r)
15             (display "\nFAILED WITH: ")
16             (display tr)
17             (newline)))))
18    ((test u (t1 r1) (t2 r2) ...)
20     (begin
21       (test u (t1 r1))
22       (test u (t2 r2) ...)))))
```

1. *Does it matter for this example that Scheme's macros are hygienic? Why? Under which circumstances would your assessment change?*
2. *Why is* `t` *evaluated to* `tr` *in the let expression in line 6 instead of simply using* `t` *in place of* `tr` *in lines 7 and 16? Can you imagine (or even show) how you instructor was forced to introduce this let expression? (Hint: Look at the test of* `substitute-fresh-vars-rec`*!) Which buzzword does apply again?*

Answer:

(a) No, because if `tr` was not uniquely renamed it could still not capture any variable since no expression is being evalutated in the scope of `tr`'s binding `let`. Subtle point: This includes the evaluation of `t` proper since the macro uses a `let` and not a `letrec` to declare `tr`. (Remember that the bound expressions of a `let`, in this case the expression subsituted for `t` are evaluated in the scope *outside* of the surrounding `let` form.) (5 Points)

The assessment would change, for example, if any expression passed to the `test` macro would be evaluated within the `let` form. If such an expression used a variable named `tr` that use could be captured by an unhygienic macro. (5 Points)

(b) Binding the result of test `t` to `tr` corresponds to a single evaluation of the expression substituted for `t`. Otherwise `t` would be evaluated a second time in line 16 in case that the test fails. This can be of importance if the evaluation of `t` has *side-effects*. (10 Points)

The instructor must have had a failure in the first test case of the `substitute-fresh-vars-rec` unit. `;-)` `substitute-fresh-vars-rec` has the side-effect of incrementing the global variable `*fresh-vars-counter*` and at the same time its result depends on the value of that counter. Therefore the failure of

the first test case changes the result of the second test case causing it to fail too even though it wasn't even modified. (5 Points)

So the buzzwords applying here are *side-effect* or *referential transparency*—mabe also *imperative vs. functional/declarative*. Any of these would suffice. (5 Points)