Problem Set 4

Due: Friday 23 October.

Problem 1. Consider the following versions for inequalities of the two substitution rules for equality:

$$(1) \qquad \models f \leq g \text{ implies } \models\ e[x := f] \leq e[x := g].$$

$$(2) \qquad \models e \leq g \text{ implies } \models\ e[x := f] \leq g[x := f].$$

Only one of these assertions is true. Prove the one which is true, and give a simple counterexample to the other one. You may assume from lecture the

Lemma. $[\![e[x := f]]\!]V = [\![e]\!](V[x \leftarrow [\![f]\!]V])$,

where for any function, F, and elements a, b, we define $F[a \leftarrow b]$ to be the function G such that

$$G(u) = F(u) \text{ for } u \neq a, \text{ and } G(a) = b.$$

Download the latest version of the pattern matching code and the Substitution Model interpreter for Scheme from the course website.

The file `match.scm` is a slightly updated version of the "context" pattern matcher from the file `pattern.scm` used in a previous assignment. The file `deriv-simplify-rules.scm` contains a set of rewrite rules for putting arithmetic expressions in standard sum-of-products form.

Problem 2. Load the files `match.scm` and `deriv-simplify-rules.scm` into Scheme. Simplify some example arithmetic expressions to see what happens; there are some examples at the end of `deriv-simplify-rules.scm`.

2(a) Examine the `simplify-rules` defined in `deriv-simplify-rules.scm`. Give an example showing that the order in which the rules appear in the list `simplify-rules` matters: a search for a final form by either of the procedures `one-rules-application` or `apply-rules` in `match.scm` may run forever if the rules are reordered.

Rules closer to the beginning of the `simplify-rules` list have priority in being applied by the rule application procedures. The rules really accomplish rewriting in a series of stages: first they "flatten" an expression by repeated use of distributivity, then all sums and nonnumeric products are right-associated, then products and then sums are sorted, and throughout numerical operations are performed whenever possible.

For "staged" rules like this, it's somewhat inefficient to keep one big rule list: at later stages all the higher-priority rules are tried first even though they will no longer be applicable. So there can be a small gain in efficiency if the rules are grouped to reflect the intended stages of the simplification process, and then final forms are found successively for each group.

A more significant gain in efficiency can be achieved by noticing that final forms for several rule-groups can be found using the `one-topdown-final-form` or `one-bottomup-final-form` procedures, instead of the very inefficient `one-everywhere-final-form` procedure (all from the file `match.scm`).

2(b) Figure out a useful way to group the rules, and in several cases to use bottomup or topdown rewriting, to find the final form of arithmetic expressions. (Some rules may appear in several groups.) Then define a procedure

```
(define (one-arithmetic-final-form expr)
  (one-<everywhere/topdown/bottomup>-final-form
    last-rule-group
    (one-<everywhere/topdown/bottomup>-final-form
      next-to-last-rule-group
      (one-...
          .
           .
            .
          (one-<everywhere/topdown/bottomup>-final-form
            first-rule-group
            expr)))))
```

Briefly justify any uses you made of bottomup or topdown final form search in your procedure definition. Turn in some examples illustrating the correctness of your procedure – it should return the same final form as

```
(one-everywhere-final-form simplify-rules expr)
```

If possible, submit timing information comparing these two procedures on various expressions. In MIT scheme, a useful utility for timing evaluations is

```
(define (timed f . args)
  (let ((init (runtime)))
    (let ((v (apply f args)))
      (write-line (list 'time: (- (runtime) init)))
      v)))
```

Problem 3. The file `sm.scm` includes many rules for "desugaring" more general Scheme expressions into a small "kernel" of Scheme. For example `drules:quote` are two simple rules for desugaring expressions with quoted lists into expression in which quotes are only used to describe symbols.

With these quote desugaring rules, the `one-everywhere-final-form` procedure is not only inefficient, but depends for its correct handling of quoted expressions on the details of how the `match` procedure searches for context-pattern matches. So it would not work with several correct, alternative versions of a `match` procedure.

3(a) Give a simple example of an expression `expr` such that

```
(one-bottomup-final-form drules:quote expr)
```

incorrectly desugars `expr`.

3(b) Explain why

```
(one-topdown-final-form drules:quote expr)
```

will correctly desugar the quotes from `expr`.