MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Compvter Science

Problem Set 7

Problem 1: Pattern Matching

This problem should be solved in the pattern-matching framework discussed in class.

a. Write a definition in Scheme for the deconstructor `cons~`.

b. Write a definition in Scheme for the deconstructor `odd~`. Inside `match`, `odd` will be used as a "constructor" taking one argument. An `(odd` P`)` pattern succeeds if P matches an integer, and that integer is odd; it fails otherwise. (Note that `odd` is not actually a constructor.)

c. Desugar the following `match` expression using the desugaring discussed in class.

```
(match e
  ((cons #f (cons #t y)) (cons #t e))
  (_ (error "Louis, get out of the kitchen!")))
```

Problem 2: Costs

Sam Antics has a new idea for a type system that is intended to help programmers estimate the running time of their programs. His idea is to develop a set of static rules that will assign every expression a *cost* as well as a type. The cost of an expression is a conservative estimate of how long the expression will take to evaluate.

Sam has developed a new language, called Discount, that uses his cost model. Discount is a call-by-value, statically typed functional language with type reconstruction. Discount is based on SCHEME/R, and inherits its types, with one major difference: a function type in Discount includes the *latent cost* of the function, that is, the cost incurred when the function is called on some arguments.

For example, the Discount type `(-> 4 (int int) int)` is the type of a function that takes two `int`s as arguments, returns an `int` as its result, and has cost at most 4 every time it is called.

The grammar of Discount is a subset of SCHEME/R, with function types extended to include costs:

$T \in$ Type
$O \in$ Primop $= \{$`+`, `-`, `*`, `/`, `<`, `=`$\}$
$E \in$ Exp
$L \in$ Lit $=$ Boollit $\cup$ Intlit
$I \in$ Identifier
$C \in$ Cost

$E ::=$ L
| I
| `(if` E_{test} E_{con} E_{alt}`)`
| `(primop` O E_{arg}`*)`
| `(lambda (`I`*)` E_{body}`)`

| (*E*$_{rator}$ *E*$_{rand}$*)
| (let ((*I* *E*)*) *E*$_{body}$)
| (letrec ((*I* *E*)*) *E*$_{body}$)

C ::= loop | *I* | (sum *C**) | (max *C**) | 0 | 1 | 2 | ...

T ::= int | bool | *I* | (-> *C* (*T**) *T*$_{body}$)

Sam has formalized his system by defining type/cost rules for Discount. The rules allow judgments of the form

$$A \vdash E : T \$ C,$$

which is pronounced, "in the type environment A, expression E has type T and cost C."

For example, here are Sam's type/cost rules for literals and(non-generic) identifiers:

$$A \vdash N : \texttt{int} \$ 1$$
$$A \vdash B : \texttt{bool} \$ 1$$
$$A[I : T] \vdash I : T \$ 1$$

That is, Sam assigns both literals and identifiers a cost of 1. In addition:

- The cost of a `lambda` expression is 2.
- The cost of an `if` expression is 1 plus the cost of the predicate expression plus the maximum of the costs of the consequent and alternate.
- The cost of an N argument application is the sum of the cost of the operator, the cost of each argument, the latent cost of the operator, and N.
- The cost of an N argument primop application is the sum of the cost of each argument, the latent cost of the primop, and N. The latent cost of the primop is determined by a *signature* Σ, a function from primop names to types. For example,

$$\Sigma(\texttt{+}) = \texttt{(-> 1 (int int) int)}.$$

Here are some example judgments that hold in Sam's system:

$$A \vdash \texttt{(primop + 2 1)} : \texttt{int} \$ 5$$
$$A \vdash \texttt{(primop + (primop + 1 2) 4)} : \texttt{int} \$ 9$$
$$A \vdash \texttt{(primop + 2 ((lambda (y) (primop + y 1)) 3))} : \texttt{int} \$ 13$$

Loop is the cost assigned to expressions that may diverge. For example, the expression

```
(letrec ((my-loop (lambda () (my-loop))))
  (my-loop))
```

is assigned cost `loop` in Discount. Because it is undecidable whether an arbitrary expression will diverge, we cannot have a decidable type/cost system in which *exactly* the diverging expressions have cost `loop`. We will settle for a system that makes a *conservative approximation*: every program that diverges will be assigned cost `loop`, but some programs that do not diverge will also be assigned `loop`.

Because Discount has non-numeric costs, like `loop` and cost identifiers (which we won't discuss), it is not so simple to define what we mean by statements like "the cost is the sum of the costs of the arguments. . . ." That is the purpose of the costs (sum C_1 C_2) and (max C_1 C_2). Part of Sam's system ensures that `sum` and `max` satisfy sensible cost equivalent axioms, such as the following:

$$\begin{aligned} \texttt{(sum } N_1 \ N_2\texttt{)} &= N_1 + N_2 \\ \texttt{(sum loop } N\texttt{)} &= \texttt{loop} \\ \texttt{(sum } N \texttt{ loop)} &= \texttt{loop} \end{aligned}$$

$$
\begin{array}{rcl}
\texttt{(sum loop loop)} & = & \texttt{loop} \\
\texttt{(max } N_1\ N_2\texttt{)} & = & \text{the max of } N_1 \text{ and } N_2 \\
\texttt{(max loop } N\texttt{)} & = & \texttt{loop} \\
\texttt{(max } N\ \texttt{loop)} & = & \texttt{loop} \\
\texttt{(max loop loop)} & = & \texttt{loop}
\end{array}
$$

You do not have to understand the details of how cost equivalences are proved in order to solve this problem.

a. Give a type/cost rule for `lambda`.

b. Give a type/cost rule for application.

c. Give a type/cost rule for `if`.

d. We would like to be able to write Discount expressions such as

```
(if w
    (lambda (x) (primop + x x))
    (lambda (y) (primop + (primop + y y) y)))
```

However, the rule for `if` requires the *types* (not costs) of the consequent and alternative of an `if` expression be identical. In this case, the types are `(-> 5 (int) int)` and `(-> 9 (int) int)`, which differ, but only in the latent cost. We can use subtyping to get around this problem.

Assume that you have a predicate `cost-leq` on costs, which works as expected:

- `(cost-leq` N_1 N_2`)` is true if and only if $N_1 \leq N_2$;
- `(cost-leq loop` N`)` is false;
- `(cost-leq` N `loop)` is true;
- `(cost-leq loop loop)` is true;

and so on.

Use `cost-leq` to give a subtyping rule for function types. (Your rule, along with the usual [*inclusion*] typing rule, should enable the system to deduce that the consequent and alternative of the example expression above have the same type.)