Problem Set 6 Solutions

Problem 1 Pattern Matching
This problem should be solved in the pattern-matching framework from Chapter 10.

a. Write a definition in FL for the deconstructor cons~.

Solution:

```fl
(define (cons~ e s f) ; s is success continuation, f is failure continuation
  (if (pair? e)
      (s (fst e) (snd e))
      (f)))
```

b. Write a definition in FL 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.)

Solution:

```fl
(define (odd~ e s f)
  (if (scand (int? e) (= (% e 2) 1))
      (s e)
      (f)))
```

c. Desugar the following match expression using the desugaring discussed in Chapter 10.

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

Solution: Here is the desugaring, in all its twisted glory. Some notes (which you can see from the desugaring rules, but are visible more concretely here): Later clauses are placed inside the failure continuations for earlier ones. No code corresponding to “don’t care” patterns (_) appears in the desugaring. User code is dwarfed by desugaring code. The desugaring is pretty inefficient—take a close look, for example, at the innermost failure clause (the one with (error no-pattern-matches!)).

Code copied from the user’s match expression is in black; all other code is in grey.
(let ((Itop e))
  (let ((Ifail1 (abs ()
    (let ((Ifail2 (abs () (error no-pattern-matches!))))
      ;; Second clause:
      (error "Louis, get out of the kitchen!")))))) ; end Ifail1
  ;; First clause:
  (cons^ Itop (abs (car1 cdr1)
    (if (equal? #f car1)
      (cons^ cdr1 (abs (car2 cdr2)
        (if (equal? #t car2)
          (let ((y cdr2)) (cons #t e))
          (Ifail1)))
        Ifail1))
    (Ifail1)))
  Ifail1)))

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 \textit{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 FL, and inherits its types, with one major difference: a function type in Discount includes the \textit{latent cost} of the function, that is, the cost incurred when the function is called on some arguments.

For example, the Discount type $\rightarrow 4 \text{(int int int)}$ is the type of a function that takes two \text{int}s as arguments, returns an \text{int} as its result, and has cost at most 4 every time it is called.

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

\[
\begin{align*}
T & \in \text{Type} \\
O & \in \text{Primop} = \{*, -, *, /, <, =\} \\
E & \in \text{Exp} \\
L & \in \text{Lit} = \text{BoolLit} \cup \text{IntLit} \\
I & \in \text{Ident} \\
C & \in \text{Cost}
\end{align*}
\]

\[
E ::= L \\
| I \\
| (if $E_{test}$ $E_{con}$ $E_{alt}$) \\
| (prim $O$ $E_{arg}$) \\
| (abs $(I^*)$ $E_{body}$) \\
| ($E_{rotor}$ $E_{rand}$) \\
| (let ($(IE)^*$) $E_{body}$) \\
| (letrec ($(IE)^*$) $E_{body}$)
\]
$C := \text{loop} \mid I \mid (\text{sum} \ C) \mid (\text{max} \ C) \mid 0 \mid 1 \mid 2 \mid \ldots$

$T := \text{int} \mid \text{bool} \mid I \mid (\to \ C \ (T)) \ T_{\text{body}}$

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:

$$A \vdash NT : \text{int} \ C 1$$
$$A \vdash B : \text{bool} \ C 1$$
$$A[I : T] \vdash I : T \ C 1$$

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

- The cost of a $\text{abs}$ expression is 2.

- The cost of an $\text{if}$ 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 prim application is the sum of the cost of each argument, the latent cost of the primop, and $N$. The latent cost of the primitive operation is determined by a signature $\Sigma$, a function from primop names to types. For example,

$$\Sigma(+) = (\to \ 1 \ (\text{int} \ \text{int}) \ \text{int}).$$

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

$$A \vdash \text{prim} + 2 1 : \text{int} \ C 5$$
$$A \vdash \text{prim} + (\text{prim} + 1 2) 4 : \text{int} \ C 9$$
$$A \vdash \text{prim} + 2 \ ((\text{abs} \ (y) \ (\text{prim} + y 1)) 3) : \text{int} \ C 13$$

$\text{loop}$ is the cost assigned to expressions that may diverge. For example, the expression

$$(\text{letrec} ((\text{my-loop} \ (\text{abs} \ () \ (\text{my-loop}))))$$

(my-loop))

is assigned cost $\text{loop}$ 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 $\text{loop}$. We will settle for a system that makes a conservative approximation: every program that diverges will be assigned cost $\text{loop}$, but some programs that do not diverge will also be assigned $\text{loop}$.

Because Discount has non-numeric costs, like $\text{loop}$ 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 $(\text{sum} \ C_1 \ C_2)$ and $(\text{max} \ C_1 \ C_2)$. Part
of Sam’s system ensures that \textit{sum} and \textit{max} satisfy sensible cost equivalent axioms, such as the following:

\[
\begin{align*}
(\text{sum } NT_1, NT_2) &= NT_1 + NT_2 \\
(\text{sum } \text{loop } NT) &= \text{loop} \\
(\text{sum } NT \text{ loop}) &= \text{loop} \\
(\text{sum } \text{loop } \text{loop}) &= \text{loop} \\
(\text{max } NT_1, NT_2) &= \text{the max of } NT_1 \text{ and } NT_2 \\
(\text{max } \text{loop } NT) &= \text{loop} \\
(\text{max } NT \text{ loop}) &= \text{loop} \\
(\text{max } \text{loop } \text{loop}) &= \text{loop}
\end{align*}
\]

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 \textit{abs}.

\textbf{Solution:}

\[
\frac{A[I_1 : T_1, \ldots, I_n : T_n] \vdash E : T \$ C}{A \vdash \text{abs}(I_1 \ldots I_n)E : (\rightarrow C (T_1 \ldots T_n) T) \$ 2} \quad \text{[abs]}
\]

b. Give a type/cost rule for application.

\textbf{Solution:}

\[
\frac{A \vdash E_0 : (\rightarrow C_{\text{latent}} (T_1 \ldots T_n) T) \$ C_0}{A \vdash \forall i.A \vdash E_i : T_i \$ C_i}
\frac{A \vdash (E_0 E_1 \ldots E_n) : T \$ (\text{sum } n C_0 C_1 \ldots C_n C_{\text{latent}})}{A \vdash (E_0 E_1 E_2) : T \$ (\text{sum } 1 C_0 (\text{max } C_1 C_2))} \quad \text{[application]}
\]

We’ve extended \textit{sum} to more than two arguments in the obvious way.

c. Give a type/cost rule for \textit{if}.

\textbf{Solution:}

\[
\frac{A \vdash E_0 : \text{bool} \$ C_0}{A \vdash E_1 : T \$ C_1}
\frac{A \vdash E_2 : T \$ C_2}{A \vdash (\text{if } E_0 E_1 E_2) : T \$ (\text{sum } 1 C_0 (\text{max } C_1 C_2))} \quad \text{[if]}
\]

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

\[
(\text{if } w)
\begin{align*}
(\text{abs } (x) \ (\text{prim } + x \ x)) \\
(\text{abs } (y) \ (\text{prim } + (\text{prim } + y \ y) \ y)))
\end{align*}
\]

4
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 \((-\rightarrow 5 \text{ int} \text{ int})\) and \((-\rightarrow 9 \text{ int} \text{ int})\), which differ, but only in the latent cost. We can use subtyping to get around this problem.

Assume that you have a predicate \(\text{cost-leq}\) on costs, which works as expected:

- \((\text{cost-leq } N T_1 \ NT_2)\) is true if and only if \(NT_1 \leq NT_2\);
- \((\text{cost-leq } \text{loop } NT)\) is false;
- \((\text{cost-leq } NT \ \text{loop})\) is true;
- \((\text{cost-leq } \text{loop } \text{loop})\) is true;

and so on.

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

**Solution:**

\[
\frac{T'_i \sqsubseteq T_i \quad T_i \sqsubseteq T'_i \quad (\text{cost-leq } C \ C'' \quad (-\rightarrow C (T_1 \ldots T_n) \ T_r) \sqsubseteq (-\rightarrow C' (T'_1 \ldots T'_n) \ T'_r)}{(-\rightarrow C' (T_1 \ldots T_n) \ T_r) \sqsubseteq (-\rightarrow C' (T'_1 \ldots T'_n) \ T'_r)}
\]