CS 565, Jens Palsberg Oct 21, 1998

Midterm Exam

Submit your solutions on separate paper (don't submit this handout). Write your name and id number on the top of the first page you submit. I will stable your pages together. This exam is intended to take you 2 hours. Budget your time wisely. There are 4 questions on the exam. Each question is worth 5 points, for a total of 20 points maximum. Write the solutions for different questions on different sheets of paper. Make clear what problem each solution is for. I have spare paper, just ask for it.

1. [**Beta-reduction**] Consider these Scheme definitions:

```
(define pa
  (lambda (x) (lambda (y) (lambda(c) ((c x) y)))))

(define fs
  (lambda (p) (p (lambda (a) (lambda (b) a)))))

(define i
  (lambda (x) ((fs ((pa fs) 5)) ((pa 6) (lambda (z) (z x))))))
```

What is produced by following two expressions?

```
(i 2)
((i 2) (lambda (x) x))
```

Justify your answer by showing a step-by-step computation.

2. [Type Soundness] Below is the grammar for a subset of Scheme.

$$
\begin{array}{lcl}
e & \in & \mathit{Expression} \\
e & ::= & \texttt{(if } e_1 \; e_2 \; e_3 \texttt{)} \mid \texttt{(+ } e_1 \; e_2 \texttt{)} \mid \texttt{(zero? } e \texttt{)} \mid \texttt{\#t} \mid \texttt{\#f} \mid c \\
c & \in & \mathit{IntegerConstant}
\end{array}
$$

A *value* is either `#t`, `#f`, or an integer constant c. We use v to range over values. We use $\lceil c \rceil$ to denote the integer represented by an integer constant c. A small-step operational semantics for the language is given by the reflexive, transitive closure of the relation $\rightarrow_V$:

$$\rightarrow_V \subseteq \mathit{Expression} \times \mathit{Expression}$$

$$\texttt{(if \#t } e_2 \; e_3 \texttt{)} \rightarrow_V e_2 \tag{1}$$

$$\texttt{(if \#f } e_2 \; e_3 \texttt{)} \rightarrow_V e_3 \tag{2}$$

$$\frac{e_1 \rightarrow_V e_1'}{\texttt{(if } e_1 \; e_2 \; e_3 \texttt{)} \rightarrow_V \texttt{(if } e_1' \; e_2 \; e_3 \texttt{)}} \tag{3}$$

$$\texttt{(+ } c_1 \; c_2\texttt{)} \rightarrow_V c \qquad \texttt{(where } \lceil c_1 \rceil + \lceil c_2 \rceil = \lceil c \rceil\texttt{)} \tag{4}$$

$$\frac{e_1 \rightarrow_V e_1'}{\texttt{(+ } e_1 \; e_2 \texttt{)} \rightarrow_V \texttt{(+ } e_1' \; e_2 \texttt{)}} \tag{5}$$

$$\frac{e_2 \rightarrow_V e_2'}{\texttt{(+ } v \; e_2 \texttt{)} \rightarrow_V \texttt{(+ } v \; e_2' \texttt{)}} \tag{6}$$

$$\texttt{(zero? } c \texttt{)} \rightarrow_V \texttt{\#t} \qquad \texttt{(where } \lceil c \rceil = 0\texttt{)} \tag{7}$$

$$\texttt{(zero? } c \texttt{)} \rightarrow_V \texttt{\#f} \qquad \texttt{(where } \lceil c \rceil \neq 0\texttt{)} \tag{8}$$

$$\frac{e \rightarrow_V e'}{\texttt{(zero? } e \texttt{)} \rightarrow_V \texttt{(zero? } e' \texttt{)}} \tag{9}$$

An expression e is *stuck* if it is not a value and there is no expression e' such that $e \to_V e'$. An expression *goes wrong* if it evaluates to a stuck expression.

There are two types: int and $\mathsf{boolean}$. If e is an expression, and t is a type, then the judgment

$$\vdash e : t$$

means that e has the type t. Formally, this holds when the judgment is derivable by a finite derivation tree using the following six rules:

$$\frac{\vdash e_1 : \mathsf{boolean} \quad \vdash e_2 : t \quad \vdash e_3 : t}{\vdash \texttt{(if } e_1 \ e_2 \ e_3 \texttt{)} : t} \tag{10}$$

$$\frac{\vdash e_1 : \mathsf{int} \quad \vdash e_2 : \mathsf{int}}{\vdash \texttt{(+ } e_1 \ e_2 \texttt{)} : \mathsf{int}} \tag{11}$$

$$\frac{\vdash e : \mathsf{int}}{\vdash \texttt{(zero? } e \texttt{)} : \mathsf{boolean}} \tag{12}$$

$$\vdash \texttt{\#t} : \mathsf{boolean} \tag{13}$$

$$\vdash \texttt{\#f} : \mathsf{boolean} \tag{14}$$

$$\vdash c : \mathsf{int} \tag{15}$$

An expression e is *well typed* if and only if there exists a type t such that $\vdash e : t$.

Prove in reasonable detail that a well typed expression cannot go wrong.

3. [Structural Subtyping] Consider the type expressions generated from the grammar:

$$t \ ::= \ v \mid \top \mid \mathsf{int} \mid \mathsf{float} \mid t_1 \to t_2 \mid \mu v.(t_1 \to t_2) \qquad \text{(types)}.$$

A type expression denotes a type which is obtained by an infinite unfolding of all occurrences of $\mu v.(t_1 \to t_2)$ using the rule

$$\mu v.(t_1 \to t_2) \quad = \quad (t_1 \to t_2)[v := (\mu v.(t_1 \to t_2))].$$

Define $\Sigma = \{\top, \mathsf{int}, \mathsf{float}, \to\}$. We represent a type as a partial function from $\{0,1\}^*$ to Σ, where we use 0 to denote Left, and 1 to denote Right.

The *parity* of $\alpha \in \{0,1\}^*$ is the number mod 2 of 0's in α. The parity of α is denoted $\pi\alpha$. A string α is said to be *even* if $\pi\alpha = 0$ and *odd* if $\pi\alpha = 1$.

Let $\leq_0$ be the partial order

$$\begin{array}{lll} \to & \leq_0 & \top \\ \mathsf{int} & \leq_0 & \mathsf{float} \ \leq_0 \top \end{array}$$

on Σ, and let $\leq_1$ be its reverse

$$\begin{array}{lll} \to & \geq_1 & \top \\ \mathsf{int} & \geq_1 & \mathsf{float} \ \geq_1 \top \end{array}$$

For type representations s, t, define $s \leq t$ if and only if $s(\alpha) \leq_{\pi\alpha} t(\alpha)$ for all $\alpha \in \mathcal{D}(s) \cap \mathcal{D}(t)$.

Here, are two type expressions s, t:

$$\begin{array}{rcl} s & = & \mu u.(((u \to u) \to \top) \to \mathsf{int}) \\ t & = & (\mu v.(((\top \to \mathsf{int}) \to v) \to \mathsf{float})) \to \top \end{array}$$

Is $s \leq t$? Justify your answer.

4. [**CPS**] Here is a definition of a Scheme function `f` which has a helper function `sum`. The function `f` takes as argument a flat list of integers `l`, say, `(1 2 3 4)` but not `(1 (2 3) 4)`. The function `sum` takes as argument a flat list of integers.

```
(define sum
  (lambda (l)
    (if (null? l)
        0
        (+ (car l) (sum (cdr l))))))

(define f
  (lambda (l)
    (if (= (sum l) 5)
        5
        (if (null? l)
            1
            (+ (sum l) (f (cdr l)))))))
```

Transform `f` and `sum` to a program with these four properties:

- all calls to programmer-defined functions are in tail position,
- all occurrences of `lambda` are in definitions at the top level,
- all data are either integers, or possibly nested lists of integers, and
- all programmer-defined functions take no arguments.

Justify your answer by showing a step-by-step transformation process.