CS 320: Mock Final

Total: 100 pts

CS 320 Course Staff

Problem 1 (10 pts) *Consider the following grammar:*

$$\begin{aligned} \langle expr \rangle &::= \mathsf{if}\ \langle expr \rangle\ \mathsf{then}\ \langle expr \rangle \mid \mathsf{if}\ \langle expr \rangle\ \mathsf{then}\ \langle expr \rangle\ \mathsf{else}\ \langle expr \rangle \mid \mathsf{true} \mid \mathsf{false} \mid \langle var \rangle \\ \langle var \rangle &::= x \end{aligned}$$

Is the grammar above ambiguous? If yes, present an expression which has two distinct parse trees according to the grammar. You do not need to present the parse trees, but you should explain your reasoning.

Solution.

Solution.

Problem 2 (15 pts) *Using the evaluation rules provided in the spec of mini-project 3, provide a derivation of the evaluation for the following expression:*

```
let x = 3 in
let y = 2 in
(fun x -> x y) (fun z -> x + z)
```

Solution.

Solution.

Problem 3 (20 pts) *Let's revisit the expression from Problem 2.*

```
let x = 3 in
let y = 2 in
(fun x -> x y) (fun z -> x + z)
```

What is the type of this expression? Using the type inference rules from the spec of mini project 3, generate the set of type constraints and use those constraints to determine the type of this expression.

Solution.

Solution.

Problem 4 (20 pts) *Recall the OCaml product type written as* $\tau_1 \times \tau_2$*. Expressions of this type are formed using the tuple construct* (e_1, e_2) *where* e_1 *has type* τ_1 *and* e_2 *has type* τ_2*. Also recall the elimination form:* match e with $\mid (x, y) \to e'$*. This can be encoded in* λ*-calculus with the following rules:*

- *Type* $\tau_1 \times \tau_2$ *can be encoded as* $\forall \alpha.(\tau_1 \to \tau_2 \to \alpha) \to \alpha$
- (e_1, e_2) *can be encoded as an expression of this type:* $\lambda f : \tau_1 \to \tau_2 \to \alpha.\, f\ e_1\ e_2$
- match e with $\mid (x, y) \to e'$ *can be encoded as* $e\ (\lambda x : \tau_1.\, \lambda y : \tau_2.\, e')$

Show that this encoding is correct. To do this, we will evaluate the encoding of the match expression in λ*-calculus with* e *being* (e_1, e_2)*, i.e., we will evaluate the encoding of* match (e_1, e_2) with $\mid (x, y) \to e'$*.*

1. *Write the encoding of this expression in* λ*-calculus.*
2. *Compute the value of this encoding using the evaluation rules of* λ*-calculus and show the evaluation derivation.*
3. *Show that this is equal to the expected value of this expression evaluated in OCaml.*

Solution.

Solution.

Problem 5 (20 pts) *Recall the type inference rules for OCaml products:*

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \dashv \mathcal{C}_1, \mathcal{C}_2}\ \textsc{Pair}$$

$$\frac{\Gamma \vdash e : \tau \dashv \mathcal{C} \qquad \alpha, \beta \text{ are fresh} \qquad \Gamma, x : \alpha, y : \beta \vdash e' : \tau' \dashv \mathcal{C}'}{\Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \mid (x, y) \to e' : \tau' \dashv \tau \doteq \alpha \times \beta, \mathcal{C}, \mathcal{C}'}\ \textsc{MatchPair}$$

Recall again the encoding from the last problem:

- *Type $\tau_1 \times \tau_2$ can be encoded as $\forall \alpha.(\tau_1 \to \tau_2 \to \alpha) \to \alpha$*
- *(e_1, e_2) can be encoded as an expression of this type: $\lambda f : \tau_1 \to \tau_2 \to \alpha.\ f\ e_1\ e_2$*
- match e with $\mid (x, y) \to e'$ *can be encoded as $e\ (\lambda x : \tau_1.\ \lambda y : \tau_2.\ e')$*

1. *Derive the constraints generated from applying the type inference rules to the encoding of (e_1, e_2) in λ-calculus. How do they relate to the constraints from rule* Pair*?*
2. *Derive the constraints generated from applying the type inference rules to the encoding of* match e with $\mid$ $(x, y) \to e'$ *in λ-calculus. How do they relate to the constraints from rule* MatchPair*?*

Please refer to the type inference rules from spec of mini project 3 for this problem.

Solution.

Solution.

Problem 6 (15 pts) *Define a function called* `permutations` *that takes a list ℓ and generates a list containing all its permutations (in any order) as output:*

$$\text{val } \texttt{permutations} : \text{`a list} \rightarrow \text{`a list list}$$

e.g. `permutations` *of* $[1; 2; 3]$ *are* $[1; 2; 3], [1; 3; 2], [2; 1; 3], [2; 3; 1], [3; 1; 2], [3; 2; 1]$. *You can assume all elements of the input list are distinct.*

Solution.

Solution.