1 OCaml Basics

- Suppose we have a type intlist defined as:

  ```ocaml
type intlist = Nil | Cons of int * intlist
  ```

  Write an OCaml function `prefix : intlist -> intlist -> bool` such that `prefix l1 l2` returns `true` if `l1` is a prefix of `l2`, that is, `l2` is `l1` followed by zero or more additional elements. For instance, `prefix (Cons (1, Cons (2, Nil))) (Cons (1, Cons (2, Cons (3, Nil))))` should return `true`, while `prefix (Cons (1, Cons (3, Nil))) (Cons (1, Cons (2, Cons (3, Nil))))` should return `false`.

2 BNF Grammars and ASTs

Consider the following BNF grammar:

```
C ::= c
B ::= bB | bC
S ::= aB
```

- Write OCaml datatypes representing the ASTs for `C`, `B`, and `S`.

- Write the OCaml value corresponding to the AST for the term abbbc, or draw the AST itself.
3 Imperative Programming

These questions refer to the type system and operational semantics in Appendices A and B.

• Given a type context $\Gamma$ such that $\Gamma(x) = \text{int}$, is the term
  
  $\text{if } x > 0 \text{ then } x := 3 \text{ else } x := x + 1$

  type-correct? Why or why not?

• Write the proof tree for the typing judgement described above. If the term is not type-correct, indicate the place in the proof tree that is impossible to complete.

• Write the next configuration that
  
  $(\text{if } x > 0 \text{ then } x := 3 \text{ else } x := x + 1, \{x = 0\})$

  steps to.

• Write the proof tree for the step above.
• Suppose you were writing an interpreter function `step_cmd : cmd -> state -> (cmd * state) option` that implements the semantics of the language. Fill in the two cases for the rules of the sequence command `c1 ; c2`.

```ocaml
let rec step_cmd (c : cmd) (s : state) : (cmd * state) option =
  match c with
  | Seq (Skip, c2) ->
  | Seq (c1, c2) ->
```

4  Object-Oriented Programming

These questions refer to the operational semantics in Appendix C and the following class definitions:

```ocaml
class Shape extends Object{
  int area;

  int getArea(){
    return this.area;
  }
}
class Square extends Shape{
  int side;

  int getArea(){
    return this.side * this.side;
  }
}
```

• Write the next configuration that

$$(x = s.getArea(), nil, \{s = r_1, t = r_2\}, \{r_1 \mapsto \text{new Square}(0,3), r_2 \mapsto \text{new Shape}(2)\})$$

steps to.

• Write the proof tree for the step above.
5 Lambda Calculus

• Write the lambda calculus term corresponding to a function that takes two arguments, and applies the second argument to itself.

• In each of the following terms, rename variables so that each variable has a unique name, and then evaluate the term using call-by-value semantics.

  – \((\lambda x. \lambda y. (y \ x)) \ (\lambda x. \lambda y. x)\)

  – \((\lambda x. (x \ y)) \ (\lambda z. (z \ x))\)

  – \(\lambda x. \lambda y. \lambda y. ((y \ y) \ x)\)

  – \((\lambda x. x) \ (\lambda x. x)\) \((\lambda x. x) \ (\lambda x. x)\)

• Write the type of each of the following terms in the simply-typed lambda calculus.

  – \(\lambda x : \text{int}. \lambda y : \text{int}. x\)

  – \(\lambda y : \text{int}. \lambda f : \text{int} \to \text{int}. \lambda x : \text{int}. (f \ y)\)

  – \(\lambda f : \text{int} \to \text{int}. \lambda x : \text{int}. \lambda y : \text{int}. f\)

  – \((\lambda x : \text{int}. \lambda f : \text{int} \to \text{int}. x) \ 10\)

6 Functional Programming

• The big-step semantics of a simple functional language with sum and tuple types is given in Appendix D.

  – What value does the following term evaluate to?

    \[
    \text{match} \ \text{fst} \ (\text{snd} \ (1, \text{inr} \ 2, \ 3)) \ \text{with} \ \text{inl} \ i \to \text{inr} \ i \ | \ \text{inr} \ j \to \text{inl} \ j
    \]
- Construct a proof tree that justifies your answer to the previous question.

- Consider the following OCaml program:

```ocaml
let m = true;;
let n = false;;
let o = true;;
let f m = if (m = n) then m else o;;
let n = true;;
f n
```

- What value does the program return?

- What is the value of `f`?

- Suppose you were implementing an evaluation function `eval : exp -> env -> value option` for the simple functional language. Fill in the case for `App` by translating the rule for application into OCaml code. You may assume the existence of a function `update` that takes an environment `ρ`, a variable `x`, and a value `v` and computes `ρ[x ↦ v]`. You may also assume that the closure `(fun x -> e, r)` is represented by the constructor `Closure ("x", e, r)

```ocaml
let rec eval (e : exp) (r : env) : value option :=
  match e with
  | App (e1, e2) ->
```
7 Type Inference and Unification

Consider the term \( \text{let } f = (\text{fun } x \rightarrow \text{fun } y \rightarrow y) \text{ in } ((f \ (\text{fun } x \rightarrow x)) \ 4) \).

- What is the type of this term?

- What is the polymorphic type assigned to \( f \)?

- The type inference rules for a simple functional language are given in Appendix \( \text{E} \). Write a proof tree showing the type (with type variables and constraints) of the term above.
The unification algorithm takes a set of constraints (equations between terms with variables) and comes up with a unifying substitution if one exists, by applying the following rules repeatedly to the set of constraints:

1. Discard: if a constraint is of the form $t = t$, discard it.
2. Substitute (L): if a constraint is of the form $x = t$, where $x$ is a variable, add $x \mapsto t$ to the current substitution, and replace $x$ by $t$ in the current substitution and the remaining constraints.
3. Substitute (R): if a constraint is of the form $t = x$, where $x$ is a variable, do the same as in the previous rule.
4. Decompose: if a constraint is of the form $f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n)$, where $f$ is a constructor (e.g., the $\rightarrow$ type), replace it with the series of constraints $t_1 = u_1, \ldots, t_n = u_n$.

Perform unification on the set of constraints produced in the previous problem. For full credit, show each step.

8 Concurrency

The semantics of a simple concurrent language are given in Appendix F.

- What are all the possible next steps that the configuration

$$(x := 3 \ || \ x := 4) \ || \ y := 1, \{x = 0, y = 0\}$$

can take?
Suppose we wanted to extend the language with a command `wait` that only steps when two threads are both trying to execute `wait`. In this case, the threads both execute and step to the following command. For instance, the only possible result of the program

\[
x := 0; y := 0; ((x := 1; wait; a := y) || (y := 2; wait; b := x))
\]

should be the environment where \(a\) is 2 and \(b\) is 1.
A Typing Rules for a Simple Imperative Language

\[
\begin{array}{ccc}
(n \text { is a number}) & (b \text { is a boolean}) & (\Gamma(x) = \tau) \\
\Gamma \vdash n : \text{int} & \Gamma \vdash b : \text{bool} & \Gamma \vdash x : \tau \\
\hline
\Gamma \vdash e_1 : \text{int} & \Gamma \vdash e_2 : \text{int} & \Gamma \vdash e_1 \oplus e_2 : \text{int} \quad \text{where } \oplus \text { is an arithmetic operator} \\
\Gamma \vdash e_1 : \text{int} & \Gamma \vdash e_2 : \text{int} & \Gamma \vdash e_1 \odot e_2 : \text{bool} \quad \text{where } \odot \text { is a boolean operator} \\
\Gamma \vdash e_1 : \text{bool} & \Gamma \vdash e_2 : \text{bool} & \Gamma \vdash e_1 \oplus e_2 : \text{bool} \\
\hline
\Gamma \vdash e_1 : \tau & \Gamma \vdash e_2 : \tau & \Gamma \vdash e_1 = e_2 : \text{bool} \\
\Gamma \vdash e_1 = e_2 : \text{bool} & \Gamma \vdash e : \text{tau} & \Gamma \vdash \text{if } e \text { then } e_1 \text { else } e_2 : \text{tau} \\
\hline
\Gamma \vdash e : \text{bool} & \Gamma \vdash c_1 : \text{ok} & \Gamma \vdash c_2 : \text{ok} \\
\Gamma \vdash \text{if } e \text { then } c_1 \text { else } c_2 : \text{ok} & \Gamma \vdash e : \text{bool} & \Gamma \vdash c : \text{ok} \\
\Gamma \vdash \text{while } e \text { do } c : \text{ok} \\
\end{array}
\]

B Operational Semantics for a Simple Imperative Language

\[
\begin{array}{ccc}
(n \text { is a number}) & (b \text { is a boolean}) & (\sigma(x) = v) \\
(n, \sigma) \Downarrow n & (b, \sigma) \Downarrow b & (x, \sigma) \Downarrow v \\
\hline
(e_1, \sigma) \Downarrow v_1 & (e_2, \sigma) \Downarrow v_2 & (v_1 \oplus v_2 = v) \\
(e_1 \oplus e_2, \sigma) \Downarrow v \\
\hline
(e, \sigma) \Downarrow \text{true} & (e_1, \sigma) \Downarrow v & (e, \sigma) \Downarrow \text{false} & (e_2, \sigma) \Downarrow v \\
\text{if } e \text { then } e_1 \text { else } e_2, \sigma \Downarrow v & \text{if } e \text { then } e_1 \text { else } e_2, \sigma \Downarrow v \\
\hline
(e, \sigma) \Downarrow v & (x := e, \sigma) \rightarrow (\text{skip}, \sigma[x \mapsto v]) \\
\hline
(c_1, \sigma) \rightarrow (c_1', \sigma') & (c_1; c_2, \sigma) \rightarrow (c_1', c_2, \sigma') \\
\text{skip}; c_2, \sigma \rightarrow (c_2, \sigma) \\
\hline
(e, \sigma) \Downarrow \text{true} & (e, \sigma) \Downarrow \text{false} \\
\text{if } e \text { then } c_1 \text { else } c_2, \sigma \rightarrow (c_1, \sigma) & \text{if } e \text { then } c_1 \text { else } c_2, \sigma \rightarrow (c_2, \sigma) \\
\text{while } e \text { do } c, \sigma \rightarrow (\text{if } e \text { then } (c; \text{while } e \text { do } c) \text { else } \text{skip}, \sigma) \\
\end{array}
\]
C  Operational Semantics for a Simple Object-Oriented Language

\[
\begin{align*}
(n &\text{ is an integer literal}) & \quad (b &\text{ is a boolean literal}) & \quad (\rho(x) = v) \\
(n, \rho, \sigma) &\downarrow n & (b, \rho, \sigma) &\downarrow b & (x, \rho, \sigma) &\downarrow v
\end{align*}
\]

\[
\begin{align*}
(e_1, \rho, \sigma) &\downarrow v_1 & (e_2, \rho, \sigma) &\downarrow v_2 & (v_1 + v_2 = v) & \quad \text{where} \oplus \text{is an arithmetic operator}
\end{align*}
\]

\[
\begin{align*}
(e, \rho, \sigma) &\downarrow r & (\sigma(r) = \text{new } C(v_1, \ldots, v_n)) & \quad (\text{fields}(C)[i] = \tau f) & \quad (e.f, \rho, \sigma) &\downarrow v_i
\end{align*}
\]

\[
\begin{align*}
(e, \rho, \sigma) &\downarrow v & ((x = e;), k, \rho, \sigma) &\rightarrow \text{(skip, } k, \rho[x \mapsto v], \sigma)
\end{align*}
\]

\[
\begin{align*}
(c_1, k, \rho, \sigma) &\rightarrow (c_1', k', \rho', \sigma') & \quad (c_1, c_2, k, \rho, \sigma) &\rightarrow (c_1', c_2', k', \rho', \sigma') & \quad (\text{skip } c_2, k, \rho, \sigma) &\rightarrow (c_2, k, \rho, \sigma)
\end{align*}
\]

\[
\begin{align*}
(e, \rho, \sigma) &\downarrow v_1 \ldots v_n & (e, \rho, \sigma) &\downarrow v' & (r \notin \text{dom}(\sigma))
\end{align*}
\]

\[
\begin{align*}
(x = \text{new } C(v_1, \ldots, v_n); k, \rho, \sigma) &\rightarrow \text{(skip, } k, \rho[x \mapsto r], \sigma[r \mapsto \text{new } C(v_1, \ldots, v_n)]) & \quad (e.f = e_1; k, \rho, \sigma) &\rightarrow \text{(skip, } k, \rho, \sigma[r \mapsto \text{new } C(v_1, \ldots, v_n)])
\end{align*}
\]

\[
\begin{align*}
(x = e.m(e_1, \ldots, e_n); k, \rho, \sigma) &\rightarrow (e, ((\rho, x) :: k), \{\text{this } = r, x_1 = v_1, \ldots, x_n = v_n\}, \sigma)
\end{align*}
\]

\[
\begin{align*}
(x = e; (\rho_0, x) :: k, \rho, \sigma) &\rightarrow \text{(skip, } k, \rho_0[x \mapsto v], \sigma)
\end{align*}
\]

D  Big-Step Operational Semantics for a Simple Functional Language

\[
\begin{align*}
(n &\text{ is an integer literal}) & \quad (b &\text{ is a boolean literal}) & \quad (\rho(x) = v) \\
(n, \rho) &\downarrow n & (b, \rho) &\downarrow b & (x, \rho) &\downarrow v
\end{align*}
\]

\[
\begin{align*}
(e_1, \rho) &\downarrow v_1 & (e_2, \rho) &\downarrow v_2 & (v_1 + v_2 = v) & \quad \text{where} \oplus \text{is an arithmetic or boolean operator}
\end{align*}
\]

\[
\begin{align*}
(\text{fun } x \rightarrow e, \rho) &\downarrow (\text{fun } x \rightarrow e, \rho) & (e_1, \rho) &\downarrow v & (e_2, \rho) &\downarrow v & (e, \rho'[x \mapsto v_2]) &\downarrow v
\end{align*}
\]

\[
\begin{align*}
(e, \rho) &\downarrow v & (\text{inl } e, \rho) &\downarrow \text{inl } v & (e, \rho) &\downarrow v & (\text{inr } e, \rho) &\downarrow \text{inr } v
\end{align*}
\]

\[
\begin{align*}
((\text{match } e \text{ with } \text{inl } x_1 \rightarrow e_1 | \text{inr } x_2 \rightarrow e_2), \rho) &\downarrow v' & (e, \rho) &\downarrow v' & (e_1, \rho[x_1 \mapsto v]) &\downarrow v'
\end{align*}
\]

\[
\begin{align*}
((\text{match } e \text{ with } \text{inl } x_1 \rightarrow e_1 | \text{inr } x_2 \rightarrow e_2), \rho) &\downarrow v' & (e, \rho) &\downarrow v' & (e_2, \rho[x_2 \mapsto v]) &\downarrow v'
\end{align*}
\]
E  Type Inference Rules for a Simple Functional Language

\[(n \text{ is an integer literal})\]
\[\frac{}{\Gamma \vdash n : \text{int} \mid \{\}}\]

\[(b \text{ is a boolean literal})\]
\[\frac{}{\Gamma \vdash b : \text{bool} \mid \{\}}\]

\[(\Gamma(x) = \tau)\]
\[\frac{}{\Gamma \vdash x : \tau \mid \{\}}\]

\[\frac{}{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2}{\Gamma \vdash e_1 + e_2 : \text{int} \mid \{\tau_1 = \text{int}, \tau_2 = \text{int}\} \cup C_1 \cup C_2}\]

\[\frac{}{\Gamma \vdash e_1 \& e_2 : \text{bool} \mid \{\tau_1 = \text{bool}, \tau_2 = \text{bool}\} \cup C_1 \cup C_2}\]

\[\frac{}{\Gamma \vdash \text{fun} x : \tau_1 \to e : \tau_1 \to \tau_2 \mid C'}\]

\[\frac{}{\Gamma \vdash \text{fun} x \to e : \tau_1 \to \tau_2 \mid C}\]

\[\frac{}{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2 \quad \tau \text{ fresh}}{\Gamma \vdash (e_1 e_2) : \tau \mid \{\tau_1 = \tau_2 \to \tau\} \cup C_1 \cup C_2}\]

\[\frac{}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau \mid C_1 \cup C_2}\]

\[\frac{}{\Gamma(x) = \forall a_1 ... a_n. \tau \quad b_1,...,b_n \text{ fresh}}{\Gamma \vdash x : \{a_1 \mapsto b_1,...,a_n \mapsto b_n\} \tau \mid \{\}}\]

where \([a_1 \mapsto b_1,...,a_n \mapsto b_n]\tau\) means “replace the bound variables \(a_1\) through \(a_n\) with the fresh type variables \(b_1\) through \(b_n\)”.

F  Operational Semantics for a Simple Concurrent Language

All of the rules of the simple imperative language, plus:

\[\frac{}{(c_1, \sigma) \to (c_1', \sigma')}{(c_1 || c_2, \sigma) \to (c_1', || c_2', \sigma')}\]

\[\frac{}{(c_2, \sigma) \to (c_2', \sigma')}{(c_1 || c_2, \sigma) \to (c_1 || c_2', \sigma')}\]

\[\frac{}{(\text{skip} || \text{skip}, \sigma) \to (\text{skip}, \sigma)}\]

\[\frac{}{(acquire(x), \sigma) \to (\text{skip}, \sigma[x \mapsto 1])}\]

\[\frac{}{(\text{release}(x), \sigma) \to (\text{skip}, \sigma[x \mapsto 0])}\]

\[\frac{}{\sigma(x) = 0}\]

\[\frac{}{\sigma(x) = 1}\]