1 Object-Oriented Languages

These questions refer to the type system and operational semantics in Appendices A and B. They also use the following class declarations:

```java
class Book extends Object {
    int pages;
    bool borrowed;
    int timesBorrowed;

    Book(int pages) {
        super();
        this.pages = pages;
    }

    int copy(Book b) {
        this.pages = b.pages;
        return this.pages;
    }
}
```

```java
class LibraryBook extends Book {
    int timesBorrowed;

    LibraryBook(int pages, bool borrowed, int timesBorrowed) {
        super(pages);
        this.borrowed = borrowed;
        this.timesBorrowed = timesBorrowed;
    }

    int borrow() {
        this.borrowed = true;
        this.timesBorrowed = this.timesBorrowed + 1;
        return this.timesBorrowed;
    }

    int return() {
        this.borrowed = false;
        return 0;
    }
}
```

- Write a proof tree for the typing judgment

\[
\Gamma \vdash \text{book1} = \text{new LibraryBook}(200, \text{true}, 1); \ x = \text{book1.copy(book1)}; : \text{ok}
\]

assuming that \(\Gamma(\text{book1}) = \text{Book}\) and \(\Gamma(x) = \text{int}\).

\[
\frac{
\Gamma \vdash 200 : \text{int} \quad \Gamma \vdash \text{true} : \text{bool} \quad \Gamma \vdash 1 : \text{int}
}{
\Gamma \vdash \text{book1} = \text{new LibraryBook}(200, \text{true}, 1); : \text{ok} \quad T_1
}
\]

where

\[
T_1 = \frac{
\Gamma \vdash \text{book1} : \text{Book}
}{
\Gamma \vdash x = \text{book1.copy(book1)}; : \text{ok}
}
\]
• Write the next configuration that

\[(x = \text{book1.copy(book1)}; \text{nil}, \{\text{book1} = r_1\}, \{r_1 \mapsto \text{new LibraryBook}(200, \text{true}, 1)\})\]

steps to.

\[(\text{this.pages} = \text{pages}; \text{return this.pages};, \{(\text{book1} = r_1), x\}, \{\text{this} = r_1, b = r_1\}, \{r_1 \mapsto \text{new LibraryBook}(200, \text{true}, 1)\})\]

• Write a proof tree for the step above.

Let \(\sigma_0\) be \(\{r_1 \mapsto \text{new LibraryBook}(200, \text{true}, 1)\}\).

\[
\begin{array}{c}
(\text{book1}, \{\text{book1} = r_1\}, \sigma_0) \downarrow r_1 \\
\hline
(x = \text{book1.copy(book1)}; \text{nil}, \{\text{book1} = r_1\}, \sigma_0) \rightarrow \\
(\text{this.pages} = \text{pages}; \text{return this.pages};, \{(\text{book1} = r_1), x\}, \{\text{this} = r_1, b = r_1\}, \sigma_0)
\end{array}
\]

• Suppose we wanted to extend our language with an expression \(e\ of\ C\) that returns \text{true} if \(e\) is an object belonging to the class named \(C\) (or a subclass of \(C\)), and \text{false} otherwise. Give semantic rules defining the behavior of \(e\ of\ C\).

\[
\begin{array}{c}
(e, \rho, \sigma) \downarrow r \\
\sigma(r) = \text{new } D(...) \\
D <: C
\end{array}
\quad
(e, \rho, \sigma) \downarrow r \\
\sigma(r) = \text{new } D(...) \\
D \not <: C
\]

\[
(e, \rho, \sigma) \downarrow v \\
v \text{ is not a reference}
\]

\[
(e, \rho, \sigma) \downarrow \text{false}
\]
2 Lambda Calculus

• Describe each of the following functions in English.

1. \( \lambda x. x \)
   The function that returns its argument.

2. \( \lambda x. \lambda y. x \)
   The function that takes two arguments and returns the first.

3. \( \lambda x. (x x) \)
   The function that takes an argument and applies it to itself.

4. \( \lambda x. (x z) \)
   The function that takes an argument and applies it to \( z \).

5. \( \lambda x. (x (\lambda x. x)) \)
   The function that takes an argument and applies it to the identity function (i.e., the function that returns its argument).

• In each of the following terms, rename variables as needed to create an equivalent term in which each bound variable has a unique name.

1. \( (\lambda x. x) (\lambda x. x) \)

2. \( \lambda x. (x (\lambda x. x)) \)

3. \( \lambda x. \lambda y. (x y) \)

4. \( (\lambda x. (x x)) x \)

• What are values in the lambda calculus? Give both a general answer and a specific example.

   Functions defined with lambdas, like \( \lambda x. x \).

• Fully evaluate the following term twice, once according to call-by-name semantics and once according to call-by-value semantics:

\[
((\lambda x. x) (\lambda y. (y y))) ((\lambda x. \lambda y. x) z) \rightarrow ((\lambda y. (y y)) ((\lambda x. \lambda y. x) z) \rightarrow ((\lambda x. \lambda y. x) z) \rightarrow (\lambda y. z) ((\lambda x. \lambda y. x) z) \rightarrow z
\]

\[
((\lambda x. x) (\lambda y. (y y))) ((\lambda x. \lambda y. x) z) \rightarrow (\lambda y. (y y)) ((\lambda x. \lambda y. x) z) \rightarrow (\lambda y. (y y)) (\lambda y. z) \rightarrow (\lambda y. z) (\lambda y. z) \rightarrow z
\]
The typing rules for simply-typed lambda calculus are given in Appendix C.

1. Write the type of the following term:

\[(\lambda f : \text{int} \rightarrow \text{int}. \ (f \ (f \ 4))) \ (\lambda x : \text{int}. \ x + 1)\]

\[\text{int}\]

2. Draw a proof tree for the typing judgment

\[\{\} \vdash ((\lambda f : \text{int} \rightarrow \text{int}. \ (f \ (f \ 4))) \ (\lambda x : \text{int}. \ x + 1)) : \tau\]

where \(\tau\) is your answer to the previous question.

Let \(\Gamma_1\) be \(\{f : \text{int} \rightarrow \text{int}\}\).

\[
\begin{array}{c}
\Gamma_1 \vdash f : \text{int} \rightarrow \text{int} \\
\Gamma_1 \vdash (f \ 4) : \text{int} \\
\end{array} \quad \begin{array}{c}
\Gamma_1 \vdash f : \text{int} \rightarrow \text{int} \\
\Gamma_1 \vdash 4 : \text{int} \\
\end{array} \quad \begin{array}{c}
\Gamma_1 \vdash (f \ (f \ 4)) : \text{int} \\
\end{array}
\]

\[\{\} \vdash (\lambda f : \text{int} \rightarrow \text{int}. \ (f \ (f \ 4))) : (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \quad T_2\]

\[\{\} \vdash (\lambda f : \text{int} \rightarrow \text{int}. \ (f \ (f \ 4))) \ (\lambda x : \text{int}. \ x + 1) : \text{int}\]

where

\[
T_2 = \begin{array}{c}
{\{x : \text{int}\} \vdash x : \text{int}} \\
{\} \vdash (\lambda x : \text{int} \cdot x + 1) : \text{int} \rightarrow \text{int}
\end{array} \quad \begin{array}{c}
{x : \text{int}} \vdash 1 : \text{int}
\end{array} \quad \begin{array}{c}
\{x : \text{int}\} \vdash x + 1 : \text{int}
\end{array}
\]
3 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?
    \[
    \text{fst} (\text{match } \text{inl} \ (\text{true}, 7) \ \text{with} \ \\
    | \ \text{inl} \ i \rightarrow (\text{snd} \ i, \ \text{fst} \ i) \ \\
    | \ \text{inr} \ j \rightarrow (j, \ \text{false}))
    \]

    \[
    7
    \]

  - Construct a proof tree that justifies your answer to the previous question.

    \[
    \frac{(\text{true}, \{\}) \Downarrow \text{true}}{(\text{true}, 7), \{\} \Downarrow (\text{true}, 7)}
    \]

    \[
    \frac{(\text{true}, 7), \{\} \Downarrow \text{inl} \ (\text{true}, 7)}{(\text{match } \text{inl} \ (true, 7) \ \text{with} \ \\
    | \ \text{inl} \ i \rightarrow (\text{snd} \ i, \ \text{fst} \ i) \ \\
    | \ \text{inr} \ j \rightarrow (j, \ \text{false}), \{\}) \Downarrow (7, \text{true})}{(\text{fst} \ (\text{match } \text{inl} \ (true, 7) \ \text{with} \ \\
    | \ \text{inl} \ i \rightarrow (\text{snd} \ i, \ \text{fst} \ i) \ \\
    | \ \text{inr} \ j \rightarrow (j, \ \text{false}), \{\}) \Downarrow 7}
    \]

    where

    \[
    T_3 = \frac{(\text{i}, \{\text{i} = (true, 7)\}) \Downarrow (true, 7)}{\frac{(\text{snd} \ i, \{\text{i} = (true, 7)\}) \Downarrow 7}{((\text{snd} \ i, \ \text{fst} \ i), \{\text{i} = (true, 7)\}) \Downarrow (7, \text{true})}}
    \]

- Consider the following OCaml program:

  \[
  \begin{align*}
  &\text{let } x = 5; \\
  &\text{let } y = \text{false}; \\
  &\text{let } f \ z = \text{if } y \ \text{then } z + 1; \\
  &\text{let } z = 3; \\
  &f \ x;
  \end{align*}
  \]

  - What value does the program return?

    \[
    6
    \]

  - What is the value of \(f\)?

    \[
    \langle \text{fun } z \rightarrow \text{if } y \ \text{then } z + 1, \{x = 5, y = \text{false}\} \rangle
    \]
• The big-step operational semantics for a simple functional language are given in Appendix D. Suppose we want to add pattern-matching on tuples to this language, so that we can write programs like

\[
\text{match } (1, \text{true}) \text{ with } \\
| (a, b) \to a + 1
\]

Give the semantics for expressions of the form `match e with | (x, y) \to e_1` by either making them into syntactic sugar for another expression or writing new big-step operational semantics rules.

\[
\begin{align*}
(e, \rho) \Downarrow (v_1, v_2) & \quad (e_1, \rho[x \mapsto v_1, y \mapsto v_2]) \Downarrow v \\
\overline{\quad \text{match } e \text{ with } | (x, y) \to e_1, \rho \Downarrow v} 
\end{align*}
\]

or

\[
\text{(match } e \text{ with } | (x, y) \to e_1) \Rightarrow ((\text{fun } x \to \text{fun } y \to e_1) (\text{fst } e) (\text{snd } e))
\]
A  Typing Rules for a Simple Object-Oriented Language

\[ C <: C \quad \text{\(C\text{T}(C) = \text{class } C \text{ extends } D\{\ldots\}\)} \quad C <: D \quad D <: E \]

\[ \frac{(n \text{ is an integer literal})}{\Gamma \vdash n : \text{int}} \quad \frac{(b \text{ is a boolean literal})}{\Gamma \vdash b : \text{bool}} \quad \frac{(x = \tau)}{\Gamma \vdash x : \tau} \quad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash e : \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 \vdash \tau_2}{\Gamma \vdash e : C} \]

\[ \frac{(\Gamma(x) = \tau)}{\Gamma \vdash x = e; \text{ ok}} \quad \frac{(\text{fields}(C) = \ldots, \tau_f, \ldots)}{\Gamma \vdash e : C \quad \text{fields}(C) = \ldots, \tau_f, \ldots} \]

\[ \frac{(\Gamma(x) = \tau)}{\Gamma \vdash e : C} \quad \frac{(\text{methods}(C) = \ldots, \tau_m(\tau_1 x_1, \ldots, \tau_n x_n)\{e\}, \ldots)}{\Gamma \vdash e : \tau_1 \vdash \tau_2} \quad \frac{(\Gamma(\text{ret}) = \tau)}{\Gamma \vdash \text{return } e : \text{ ok}} \]

B  Operational Semantics for a Simple Object-Oriented Language

\[ (n, \rho, \sigma) \downarrow n \quad (b, \rho, \sigma) \downarrow b \quad (x = v) \quad (\rho(x) = v) \]

\[ \frac{(e_1, \rho, \sigma) \downarrow v_1 \quad (e_2, \rho, \sigma) \downarrow v_2 \quad (v_1 \oplus v_2 = v)}{(e_1 \oplus e_2, \rho, \sigma) \downarrow v} \quad \frac{(e, \rho, \sigma) \downarrow v}{(\text{fields}(C)[i] = \tau_f)} \quad \frac{(e, \rho, \sigma) \downarrow v}{(e.f, \rho, \sigma) \downarrow v_i} \]

\[ (c_1, k, \rho, \sigma) \rightarrow (c'_1, k', \rho', \sigma') \quad (c_1, c_2, k, \rho, \sigma) \rightarrow (c'_1, c'_2, k', \rho', \sigma') \quad (\text{skip } c_2, k, \rho, \sigma) \rightarrow (c_2, k, \rho, \sigma) \]

\[ (x = \text{new } C(e_1, \ldots, e_n); k, \rho, \sigma) \rightarrow (\text{skip } k, \rho, \sigma[x \mapsto v], \sigma) \]

\[ (e, \rho, \sigma) \downarrow v \quad (e.f = e_1; k, \rho, \sigma) \rightarrow (\text{skip } k, \rho, \sigma[r \mapsto \text{new } C(v_1, \ldots, v_n)]) \]

\[ (e, \rho, \sigma) \downarrow r \quad (\sigma(r) = \text{new } C(v_1, \ldots, v_i, \ldots, v_n)) \quad (\text{fields}(C)[i] = \tau_f) \quad (e, \rho, \sigma) \downarrow v' \]

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

\[ (e, \rho, \sigma) \downarrow v \quad (\text{return } e; ((\rho_0, x) : k), \rho, \sigma) \rightarrow (\text{skip } k, \rho_0[x \mapsto v], \sigma) \]
C Typing Rules for Simply-Typed Lambda Calculus

- \( (n \text{ is an integer literal}) \quad \frac{}{\Gamma \vdash n : \text{int}} \)
- \( \frac{}{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma[x \mapsto \tau_1] \vdash e : \tau_2} \Gamma \vdash \lambda x : \tau_1. e : \tau_1 \to \tau_2 \)
- \( \frac{}{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1} \Gamma \vdash (e_1 \ e_2) : \tau_2 \)
- \( \frac{}{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1} \Gamma \vdash (e_1 \ e_2) : \tau_2 \)

D Big-Step Operational Semantics for a Simple Functional Language

- \( (n \text{ is an integer literal}) \quad \frac{}{(n, \rho) \Downarrow n} \)
- \( (b \text{ is a boolean literal}) \quad \frac{}{(b, \rho) \Downarrow b} \)
- \( (\rho(x) = v) \quad \frac{}{(x, \rho) \Downarrow v} \)
- \( (\rho(x) = v) \quad \frac{}{(x, \rho) \Downarrow v} \)

- \( (e_1, \rho) \Downarrow v_1 \quad (e_2, \rho) \Downarrow v_2 \quad (v_1 \oplus v_2 = v) \quad \text{where} \oplus \text{is an arithmetic or boolean operator} \)

- \( (\text{fun} \ x \to e, \rho) \Downarrow (\text{fun} \ x \to e, \rho) \)

- \( (e_1, \rho) \Downarrow v_1 \quad (e_2, \rho) \Downarrow v_2 \quad ((e_1, e_2), \rho) \Downarrow (v_1, v_2) \)

- \( (e, \rho) \Downarrow v \quad (\text{fst} \ e, \rho) \Downarrow v_1 \quad (\text{snd} \ e, \rho) \Downarrow v_2 \)

- \( (e, \rho) \Downarrow v \quad (\text{inl} \ e, \rho) \Downarrow \text{inl} \, v \quad (\text{inr} \ e, \rho) \Downarrow \text{inr} \, v \)

- \( ((\text{match} \ e \text{ with inl} \ x_1 \to e_1 | \text{inr} \ x_2 \to e_2), \rho) \Downarrow v' \)

- \( ((\text{match} \ e \text{ with inl} \ x_1 \to e_1 | \text{inr} \ x_2 \to e_2), \rho) \Downarrow v' \)