1 OCaml Basics

- Suppose we have a type \texttt{intlist} defined as:

\[
\texttt{type intlist = Nil | Cons of int * intlist}
\]

Write an OCaml function \texttt{avg : intlist -> int} that returns the average of all elements in a list. If the argument to \texttt{avg} is an empty list, it should return 0.

\[
\begin{aligned}
\text{let rec sum (l : intlist) : int =} \\
&\quad \text{match l with} \\
&\quad | \text{Nil} \rightarrow 0 \\
&\quad | \text{Cons (i, rest)} \rightarrow i + \text{sum rest}
\end{aligned}
\]

\[
\begin{aligned}
\text{let rec length (l : intlist) : int =} \\
&\quad \text{match l with} \\
&\quad | \text{Nil} \rightarrow 0 \\
&\quad | \text{Cons (i, rest)} \rightarrow 1 + \text{length rest}
\end{aligned}
\]

\[
\text{let avg l = sum l / length l}
\]

2 BNF Grammars and ASTs

Consider the following BNF grammar:

\[
\begin{align*}
A &::= \text{True} | \text{False} \\
C &::= \land | \lor \\
S &::= A | S \ C \ S | \neg S
\end{align*}
\]

- Write OCaml datatypes representing the ASTs for \(A\), \(S\), and \(C\).

\[
\begin{aligned}
\text{type astA = True | False} \\
\text{type astC = And | Or} \\
\text{type astS = Atom of astA | Op of astS * astC * astS | Not of astS}
\end{aligned}
\]

- Write the OCaml value corresponding to the AST for the term \(\neg((\text{True} \land \text{False}) \land (\text{False} \lor \text{True}))\), or draw the AST itself.

\[
\text{Not (Op (Op (Atom True, And, Atom False), And, Op (Atom False, Or, Atom True)))}
\]
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
  \[
  \text{while } x < 3 \text{ do } x := x + 1
  \]
  type-correct? Why or why not?
  Yes (given the updated typing rules).

- 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.

\[
\begin{array}{c}
(\Gamma(x) = \text{int}) \\
\Gamma \vdash x : \text{int} \\
\hline
(3 \text{ is a number}) \\
\Gamma \vdash 3 : \text{int} \\
\hline
(\Gamma(x) = \text{int}) \\
(\Gamma(x) = \text{int}) \\
\hline
\Gamma \vdash x + 1 : \text{int} \\
(\Gamma(x) = \text{int}) \\
\Gamma \vdash x : \text{int} \\
\begin{array}{c}
(\Gamma(x) = \text{int}) \\
\Gamma \vdash x < 3 : \text{bool} \\
\hline
(1 \text{ is a number}) \\
\Gamma \vdash 1 : \text{int} \\
\hline
\begin{array}{c}
\Gamma \vdash \text{while} \ x < 3 \text{ do } x := x + 1 \ : \text{ok}
\end{array}
\end{array}
\]

- Write the next configuration that
  \[
  ((x := \text{if } z = 2 \text{ then } 3 \text{ else } 4; \ z := x + (2 \times y)), \{z = 3\})
  \]
  steps to.

\[
((\text{skip}; \ z := x + (2 \times y)), \{z = 3, x = 4\})
\]

- Write the proof tree for the step above.

\[
\begin{array}{c}
(\{z = 3\}(z) = 3) \\
(z, \{z = 3\}) \downarrow 3 \\
\hline
(2 \text{ is a number}) \\
(2, \{z = 3\}) \downarrow 2 \\
\hline
((3 = 2) = \text{false}) \\
(3 = 2) = \text{false} \\
\hline
(4 \text{ is a number}) \\
(4, \{z = 3\}) \downarrow 4 \\
\hline
(\text{if } z = 2 \text{ then } 3 \text{ else } 4, \{z = 3\}) \downarrow 4 \\
(\text{if } z = 2 \text{ then } 3 \text{ else } 4, \{z = 3\}) \downarrow 4 \\
\hline
((x := \text{if } z = 2 \text{ then } 3 \text{ else } 4; \ z := x + (2 \times y)), \{z = 3\}) \rightarrow \\
((\text{skip}; \ z := x + (2 \times y)), \{z = 3, x = 4\})
\end{array}
\]
• Suppose we were writing an interpreter function `eval_exp : exp -> state -> value` that evaluates an expression to a value (assuming no errors). Fill in the code for evaluating the `Eq` (equals) expression.

```ocaml
let rec eval_exp (e : exp) (s : state) : value =
  match e with
  | Eq (e1, e2) -> BoolV (eval_exp e1 s = eval_exp e2 s)
```

4 Object-Oriented Programming

• The small-step semantics of function calls in a simple imperative language is given by the following rule:

\[
\begin{align*}
  (e_1, \sigma) & \Downarrow v_1 \ldots (e_n, \sigma) \Downarrow v_n \\
  \sigma(f) &= (x_1 \ldots x_n, c) \\
  (x := f(e_1, \ldots, e_n), \sigma, k) &\rightarrow (c, \{x_1 = v_1, \ldots, x_n = v_n\}, (\sigma, x) :: k)
\end{align*}
\]

A method invocation in an object-oriented language has the form `x := e.m(e_1, \ldots, e_n)`, where `e` should evaluate to an object. Instead of looking up the definition of a function `f` in the environment, the class of `e` should be used to determine which version of method `m` to invoke. Write a modified version of the rule above that gives the semantics of method invocations. You may assume the existence of any necessary helper functions.

\[
\begin{align*}
  (e, \sigma) & \Downarrow \text{new } C(...) \\
  (e_1, \sigma) & \Downarrow v_1 \ldots (e_n, \sigma) \Downarrow v_n \\
  \text{methods}(C) &= \ldots, m(x_1 \ldots x_n)\{c\}, \ldots \\
  (x := e.m(e_1, \ldots, e_n), \sigma, k) &\rightarrow (c, \{x_1 = v_1, \ldots, x_n = v_n\}, (\sigma, x) :: k)
\end{align*}
\]

• An environment mapping variables to values can be represented in OCaml with a type `env = ident -> value option`. Write a function `make_env : ident list -> value list -> env` that takes a list of identifiers `li` and a list of values `lv`, and returns an environment that maps each element of `li` to the corresponding element of `lv`. For instance, `make_env ["x", "y", "z"] [IntV 0, IntV 1, IntV 2]` should return an environment that maps `x` to 0, `y` to 1, and `z` to 2. You may assume the existence of an empty environment `empty_state`, and an `update` function of type `env -> ident -> value -> env`.

```ocaml
let rec make_env (li : ident list) (lv : value list) : env =
  match li, lv with
  | i :: li', v :: lv' -> update (make_env li' lv') i v
  | _, _ -> empty_state
```
Consider the following set of rules, defining the subtyping relation:

\[
\begin{align*}
C <: C & \quad \text{CT}(C) = \text{class } C \text{ extends } D \{\ldots\} \quad C <: D \\
C <: D & \quad C <: E
\end{align*}
\]

- Are these rules syntax-directed? Why or why not?
  No: we can’t tell by the structure of a statement of the form \( A <: B \) whether to apply the second or the third rule.

- Why is it helpful to have a syntax-directed system of rules for subtyping?
  A syntax-directed system of rules immediately leads to an algorithm for determining whether a given judgment is provable. A syntax-directed rule system for subtyping would give us an algorithm for determining whether one type is a subtype of another.

## 5 Lambda Calculus

- Write the lambda calculus terms corresponding to a function that takes two arguments, and applies the first argument to the second argument.

\[ \lambda x. \lambda y. x \, y \]

- In the following term, draw a line from each variable occurrence to the place where it is bound:

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

- Fully evaluate the following term according to call-by-value semantics:

\[ (\lambda x. x \, y) \, (\lambda y. y \, z) \, w \]

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

- Write the type of the following term in the simply-typed lambda calculus:

\[ (\lambda y : \text{int}. \, \lambda f : \text{int} \rightarrow \text{int}. \, f \, y) \, ((\lambda x : \text{int}. \, x) \, (2)) \]

\[ (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \]

## 6 Functional Programming

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

  - What value does the following term evaluate to?

  `match fst (inl 1, inr true) with
  | inl i -> i + 1
  | inr j -> if j then 3 else 4
  2`
Construct a proof tree that justifies your answer to the previous question.

<table>
<thead>
<tr>
<th>(1, ρ) ↓ 1</th>
<th>(true, ρ) ↓ true</th>
</tr>
</thead>
<tbody>
<tr>
<td>(inl 1, ρ) ↓ inl 1</td>
<td>(inr true, ρ) ↓ inr true</td>
</tr>
<tr>
<td>((inl 1, inr true), ρ) ↓ (inl 1, inr true)</td>
<td>(1, ρ[i → 1]) ↓ 1</td>
</tr>
<tr>
<td>(fst (inl 1, inr true), ρ) ↓ inl 1</td>
<td>(1 + 1, ρ[i → 1]) ↓ 2</td>
</tr>
<tr>
<td>(match fst (inl 1, inr true) with inl i → i + 1</td>
<td>inr j → if j then 3 else 4), ρ) ↓ 2</td>
</tr>
</tbody>
</table>

- Consider the following OCaml program:

```
let a = 2;;
let b = 3;;
let f x = (a * x) + b;;
let a = 4;;
f 5;;
```

- What value does the program return?
  13
- What is the value of `f`?
  The closure `(fun x -> a * x + b, {a = 2, b = 3})`

- Suppose we add an expression to our simple functional language of the form `e₁ := e₂`, which evaluates `e₁` to a reference `ℓ` and sets the value stored in `ℓ` to `e₂`. To support this, the evaluation relation will have the form `(e; ρ) ↓ (v; σ')`, where `e` is an expression, `ρ` is an environment, `σ` is a store, `v` is the value of `e`, and `σ'` is the new store created by evaluating any reference assignments in `e`.

- Write an inference rule for evaluating the addition operation in this language with left-to-right evaluation order.

```
\[
\frac{(e₁, ρ, σ) ↓ (v₁, σ₁) \quad (e₂, ρ, σ₁) ↓ (v₂, σ₂) \quad (v = v₁ + v₂)}{(e₁ + e₂, ρ, σ) ↓ (v, σ₂)}
\]
```

- Write inference rules for addition that allow either left-to-right or right-to-left evaluation.

```
\[
\frac{(e₁, ρ, σ) ↓ (v₁, σ₁) \quad (e₂, ρ, σ₁) ↓ (v₂, σ₂) \quad (v = v₁ + v₂)}{(e₁ + e₂, ρ, σ) ↓ (v, σ₂)}
\]
\[
\frac{(e₁, ρ, σ₂) ↓ (v₁, σ₁) \quad (e₂, ρ, σ₂) ↓ (v₂, σ₂) \quad (v = v₁ + v₂)}{(e₁ + e₂, ρ, σ) ↓ (v, σ₁)}
\]
The type inference rule for the if-then-else expression in an OCaml-like language is:

\[
\Gamma 
\vdash \ \text{if } e \ \text{then } e_1 \ \text{else } e_2 \ : \ \tau_1 \quad \text{if } e \ \text{then } e_1 \ \text{else } e_2 \ : \ \tau_2 \quad \text{for } \ C_1 
\vdash \ \tau = \text{bool}, \tau_1 = \tau_2 \cup C \cup C_1 \cup C_2
\]

Suppose you were implementing type inference by writing a function \texttt{get\_constraints}\( : \text{context} \rightarrow \text{exp} \rightarrow (\text{typ} \times \text{constraints})\)\ option, where a constraint \(x = y\) is represented by the pair \((x, y)\), and the constraint set is implemented as a list. Fill in the \texttt{If} case of \texttt{get\_constraints} by translating this rule into OCaml code.

```ocaml
let rec get_constraints (gamma : context) (e : exp) : (typ * constraints) option =
match e with
| If (e, e1, e2) ->
  (match get_constraints gamma e, get_constraints gamma e1, get_constraints gamma e2 with
   | Some (t, c), Some (t1, c1), Some (t2, c2) ->
     Some (t1, [(t, Tbool); (t1, t2)] @ c @ c1 @ c2)
   | _, _, _ -> None)
```

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 following set of constraints:

\[\{a = b \rightarrow c, b = \text{int} \rightarrow d, d = d, d = \text{bool} = \text{bool} \rightarrow c\}\]

For full credit, show each step.

<table>
<thead>
<tr>
<th>Step</th>
<th>Substitution</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Substitute (L) (a = b \rightarrow c)</td>
<td>({a \mapsto b \rightarrow c})</td>
<td>({b = \text{int} \rightarrow d, d = d, d = \text{bool} = \text{bool} \rightarrow c})</td>
</tr>
<tr>
<td>Substitute (L) (b = \text{int} \rightarrow d)</td>
<td>({a \mapsto (\text{int} \rightarrow d) \rightarrow c, b \mapsto \text{int} \rightarrow d})</td>
<td>({d = d, d = \text{bool} = \text{bool} \rightarrow c})</td>
</tr>
<tr>
<td>Discard (d = d)</td>
<td>({a \mapsto (\text{int} \rightarrow d) \rightarrow c, b \mapsto \text{int} \rightarrow d})</td>
<td>({d \rightarrow \text{bool} = \text{bool} \rightarrow c})</td>
</tr>
<tr>
<td>Decompose (d \rightarrow \text{bool} = \text{bool} \rightarrow c)</td>
<td>({a \mapsto (\text{int} \rightarrow d) \rightarrow c, b \mapsto \text{int} \rightarrow d})</td>
<td>({d = \text{bool}, \text{bool} = c})</td>
</tr>
<tr>
<td>Substitute (L) (d = \text{bool})</td>
<td>({a \mapsto (\text{int} \rightarrow \text{bool}) \rightarrow c, b \mapsto \text{int} \rightarrow \text{bool}, d \mapsto \text{bool}})</td>
<td>({\text{bool} = c})</td>
</tr>
<tr>
<td>Substitute (R) (\text{bool} = c)</td>
<td>({a \mapsto (\text{int} \rightarrow \text{bool}) \rightarrow \text{bool}, b \mapsto \text{int} \rightarrow \text{bool}, d \mapsto \text{bool}, c \mapsto \text{bool}})</td>
<td>({})</td>
</tr>
</tbody>
</table>
8 Logic Programming

- The core semantic rule of logic programming is as follows:

\[
\begin{align*}
  r & \in R \\
  \text{make\_fresh}(r) & = t : -t_1, ..., t_n \\
  \text{unify}(g, t) & = \sigma_1 \\
  R_0 \vdash (g :: gs, R, \sigma, k) \rightarrow ([\sigma_1][[t_1; ...; t_n] @ gs], R_0, \sigma_1 \circ \sigma, (g :: gs, R - \{r\}, \sigma) :: k)
\end{align*}
\]

This implements a depth-first proof search, in which we try to solve all the premises of the first goal before moving on to the second goal.

Write a modified version of this rule that instead implements breadth-first proof search, applying a rule to each of the current goals before moving on to the premises of the applied rules. The behavior when a rule fails to apply can remain unchanged.

\[
\begin{align*}
  r & \in R \\
  \text{make\_fresh}(r) & = t : -t_1, ..., t_n \\
  \text{unify}(g, t) & = \sigma_1 \\
  R_0 \vdash (g :: gs, R, \sigma, k) \rightarrow ([\sigma_1][[t_1; ...; t_n] @ gs], R_0, \sigma_1 \circ \sigma, (g :: gs, R - \{r\}, \sigma) :: k)
\end{align*}
\]

9 Floyd-Hoare Logic

The rules of Floyd-Hoare Logic for a simple imperative programming language are given in Appendix D.

- Construct a proof tree for the Hoare triple

\[
\begin{align*}
  \{\text{true}\} & \xRightarrow{x := 5} \{x = 5\}
\end{align*}
\]

- Construct a proof tree for the Hoare triple

\[
\begin{align*}
  \{y = 2\} & \xRightarrow{x := y + 1; z := x} \{z = 3\}
\end{align*}
\]

- Write an informative precondition and postcondition for the following program:

\[
\begin{align*}
  \{x \leq 5\} \\
  \text{while } x \leq 4 \text{ do} \\
  \{x \leq 5 \land x \leq 4\} \\
  x := x + 1 \\
  \{x \leq 5\} \\
  \{x = 5\}
\end{align*}
\]

- Annotate the program from the previous problem with conditions showing the outline of a Hoare logic correctness proof. For full credit, also show any logical implications that need to hold for the proof to be correct.

10 Concurrency

Suppose we wanted to add concurrency features to an existing language by extending its syntax with the following commands:

\[
C ::= ... \mid C \parallel C \mid \text{send}(E) \mid \langle \text{id} \rangle = \text{recv}()
\]

You may assume that the small-step relation \((c, \sigma) \rightarrow (c', \sigma')\) is already defined for the sequential part of the language.
• Give small-step semantics rules for the language such that `send` and `recv` provide synchronous message-passing.

\[
(c_1, \sigma_1) \rightarrow (c'_1, \sigma'_1)
\]
\[
(c_2, \sigma_2) \rightarrow (c'_2, \sigma'_2)
\]
\[
(c_1, \sigma_1) \parallel (c_2, \sigma_2) \rightarrow (c'_1, \sigma'_1) \parallel (c'_2, \sigma'_2)
\]

\[
(e, \sigma_1) \Downarrow v
\]
\[
(x = \text{recv}(.), \sigma_2) \rightarrow (\text{skip}, \sigma_1) \parallel (\text{skip}, \sigma_2[x \mapsto v])
\]

\[
(e, \sigma_2) \Downarrow v
\]
\[
(x = \text{recv}(.), \sigma_1) \parallel (\text{send}(e), \sigma_2) \rightarrow (\text{skip}, \sigma_1[x \mapsto v]) \parallel (\text{skip}, \sigma_2)
\]

• Give small-step semantics rules for the language such that `send` and `recv` provide asynchronous message-passing. The small-step relation for concurrent programs will need to have an extra component, a message pool \(M\).

\[
(c_1, \sigma_1, M) \rightarrow (c'_1, \sigma'_1, M')
\]
\[
((c_1, \sigma_1) \parallel (c_2, \sigma_2), M) \rightarrow ((c'_1, \sigma'_1) \parallel (c'_2, \sigma'_2), M')
\]
\[
(c_2, \sigma_2, M) \rightarrow (c'_2, \sigma'_2, M')
\]
\[
((c_1, \sigma_1) \parallel (c_2, \sigma_2), M) \rightarrow ((c_1, \sigma_1) \parallel (c'_2, \sigma'_2), M')
\]

\[
(e, \sigma) \Downarrow v
\]
\[
(x = \text{recv}(.), \sigma, v \cup M) \rightarrow (\text{skip}, \sigma[x \mapsto v], M)
\]

\[
(e, \sigma) \Downarrow v
\]
\[
(x = \text{recv}(.), \sigma, v \cup M) \rightarrow (\text{skip}, \sigma[x \mapsto v], M)
\]

11 Low-Level Languages
The semantics of a simple assembly language are given in Appendix E.

• Suppose we wanted to extend the language with a command `broff` such that `broff x, e` evaluates \(e\) to a number \(n\), and then jumps to the \(n\)th instruction after the label \(x\). For instance, in the following program, the `broff` command would jump to the assignment to \(y\), skipping the second assignment to \(x\):

\[
\begin{align*}
x &= \text{add} 1, 0 \\
\text{broff label1, x} \\
\text{label1:} \\
x &= \text{add} x, 3 \\
y &= \text{mul} x, 2
\end{align*}
\]

Write a small-step semantics rule for the `broff` command.

\[
C(p) = (\text{broff } x, e) \\
L(x) = p' \\
(e, \rho) \Downarrow v
\]
\[
C, L \vdash (p, \rho, \sigma) \rightarrow (p' + v, \rho, \sigma)
\]
A  Typing Rules for a Simple Imperative Language

\[(n \text{ is a number}) \quad (b \text{ is a boolean}) \quad (\Gamma(x) = \tau)\]

\[
\begin{array}{c}
\Gamma \vdash n : \text{int} \\
\Gamma \vdash b : \text{bool} \\
\Gamma \vdash x : \tau
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e_1 : \text{int} \\
\Gamma \vdash e_2 : \text{int}
\end{array}
\]

\[
\Gamma \vdash e_1 + e_2 : \text{int}
\]

where \(\oplus\) is an arithmetic operator

\[
\begin{array}{c}
\Gamma \vdash e_1 : \text{int} \\
\Gamma \vdash e_2 : \text{int}
\end{array}
\]

\[
\Gamma \vdash e_1 \oplus e_2 : \text{bool}
\]

where \(\oplus\) is a comparison operator

\[
\begin{array}{c}
\Gamma \vdash e_1 : \text{bool} \\
\Gamma \vdash e_2 : \text{bool}
\end{array}
\]

\[
\Gamma \vdash e_1 \odot e_2 : \text{bool}
\]

where \(\odot\) is a boolean operator

\[
\begin{array}{c}
\Gamma \vdash e_1 : \tau \\
\Gamma \vdash e_2 : \tau
\end{array}
\]

\[
\Gamma \vdash e_1 = e_2 : \text{bool}
\]

\[
\begin{array}{c}
\Gamma \vdash e : \text{bool} \\
\Gamma \vdash c_1 : \text{ok} \\
\Gamma \vdash c_2 : \text{ok}
\end{array}
\]

\[
\Gamma \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 : \text{ok}
\]

\[
\Gamma \vdash e : \text{bool} \\
\Gamma \vdash c : \text{ok}
\]

\[
\Gamma \vdash \text{while } e \text{ do } c : \text{ok}
\]

B  Operational Semantics for a Simple Imperative Language

\[(n \text{ is a number}) \quad (b \text{ is a boolean}) \quad (\sigma(x) = v)\]

\[
\begin{array}{c}
(n, \sigma) \Downarrow n \\
(b, \sigma) \Downarrow b \\
(x, \sigma) \Downarrow v
\end{array}
\]

\[
\begin{array}{c}
(e_1, \sigma) \Downarrow v_1 \\
(e_2, \sigma) \Downarrow v_2 \\
(e_1 \oplus e_2, \sigma) \Downarrow v
\end{array}
\]

\[
\begin{array}{c}
(v_1 \oplus v_2 = v)
\end{array}
\]

where \(\oplus\) is an arithmetic, comparison, or boolean operator

\[
\begin{array}{c}
(e, \sigma) \Downarrow \text{true} \\
(e_1, \sigma) \Downarrow v
\end{array}
\]

\[
\begin{array}{c}
\text{if } e \text{ then } e_1 \text{ else } e_2, \sigma \Downarrow v
\end{array}
\]

\[
\begin{array}{c}
(e, \sigma) \Downarrow \text{false} \\
(e_2, \sigma) \Downarrow v
\end{array}
\]

\[
\begin{array}{c}
\text{if } e \text{ then } e_1 \text{ else } e_2, \sigma \Downarrow v
\end{array}
\]

\[
\begin{array}{c}
(e, \sigma) \Downarrow \text{true}
\end{array}
\]

\[
\begin{array}{c}
\text{if } e \text{ then } c_1 \text{ else } c_2, \sigma \Downarrow (c_1, \sigma)
\end{array}
\]

\[
\begin{array}{c}
\text{if } e \text{ then } c_1 \text{ else } c_2, \sigma \Downarrow (c_1, \sigma)
\end{array}
\]

\[
\begin{array}{c}
\text{if } e \text{ then } e_1 \text{ else } e_2, \sigma \Downarrow (c_1, \sigma)
\end{array}
\]

\[
\begin{array}{c}
\text{if } e \text{ then } e_1 \text{ else } e_2, \sigma \Downarrow (c_1, \sigma)
\end{array}
\]

\[
\begin{array}{c}
\text{if } e \text{ then } c_1 \text{ else } c_2, \sigma \Downarrow (c_1, \sigma)
\end{array}
\]

\[
\begin{array}{c}
\text{if } e \text{ then } c_1 \text{ else } c_2, \sigma \Downarrow (c_1, \sigma)
\end{array}
\]

\[
\begin{array}{c}
\text{if } e \text{ then } e_1 \text{ else } e_2, \sigma \Downarrow (c_1, \sigma)
\end{array}
\]

\[
\begin{array}{c}
\text{if } e \text{ then } e_1 \text{ else } e_2, \sigma \Downarrow (c_1, \sigma)
\end{array}
\]

\[
\begin{array}{c}
\text{if } e \text{ then } c_1 \text{ else } c_2, \sigma \Downarrow (c_1, \sigma)
\end{array}
\]

\[
\begin{array}{c}
\text{if } e \text{ then } c_1 \text{ else } c_2, \sigma \Downarrow (c_1, \sigma)
\end{array}
\]

\[
\begin{array}{c}
\text{if } e \text{ then } e_1 \text{ else } e_2, \sigma \Downarrow (c_1, \sigma)
\end{array}
\]

\[
\begin{array}{c}
\text{if } e \text{ then } e_1 \text{ else } e_2, \sigma \Downarrow (c_1, \sigma)
\end{array}
\]

\[
\begin{array}{c}
\text{if } e \text{ then } c_1 \text{ else } c_2, \sigma \Downarrow (c_1, \sigma)
\end{array}
\]

\[
\begin{array}{c}
\text{if } e \text{ then } e_1 \text{ else } e_2, \sigma \Downarrow (c_1, \sigma)
\end{array}
\]

\[
\begin{array}{c}
\text{if } e \text{ then } c_1 \text{ else } c_2, \sigma \Downarrow (c_1, \sigma)
\end{array}
\]

\[
\begin{array}{c}
\text{while } e \text{ do } c, \sigma \Downarrow \left(\text{skip}, \sigma \left[ x \mapsto v \right]\right)
\end{array}
\]
C  Big-Step Operational Semantics for a Simple Functional Language

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

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

\[(\text{fun } x \to e, \rho) \Downarrow (\text{fun } x \to e, \rho') \quad (e_1, \rho) \Downarrow v_1 \quad (e_2, \rho \times v_2) \Downarrow v \quad (e_1, e_2, \rho) \Downarrow (v_1, v_2) \quad (e, \rho) \Downarrow v\]

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

D  Floyd-Hoare Logic for a Simple Imperative Language

\[
\{P\} \text{ skip } \{P\} \quad \{P\} \ c_1 \{Q\} \quad \{Q\} \ c_2 \{R\} \quad \{P \land (e = true)\} \ c_1 \{Q\} \\
\quad \{P \land (e = false)\} \ c_2 \{Q\} \quad \{P\} \ \text{if } e \text{ then } c_1 \text{ else } c_2 \{Q\} \quad \{P\} \ \text{while } e \text{ do } c \{P \land (e = false)\} \]

E  Small-Step Semantics for a Simple Assembly Language

\[
C(p) = (x = \text{add } e_1, e_2) \quad (e_1, \rho) \Downarrow v_1 \quad (e_2, \rho) \Downarrow v_2 \quad (v = v_1 + v_2) \quad C(p) = (x = \text{load } e) \quad (e, \rho) \Downarrow \ell \quad \sigma(\ell) = v \quad C(p) = (x = \text{store } e_1, e_2) \quad (e_1, \rho) \Downarrow \ell \quad (e_2, \rho) \Downarrow v \quad C(p) = (\text{br } x) \quad \ell(L(x)) = \rho' \quad C(p) = (\text{cbr } e, x) \quad (e, \rho) \Downarrow 0 \quad C(p) = (\text{cbr } e, x) \quad (e, \rho) \Downarrow 0 \]

\[
C, L \vdash (p, \rho, \sigma) \to (p + 1, \rho[x \mapsto v], \sigma) \quad C, L \vdash (p, \rho, \sigma) \to (p + 1, \rho, \sigma[\ell \mapsto v]) \quad C, L \vdash (p, \rho, \sigma) \to (p', \rho, \sigma) \quad C, L \vdash (p, \rho, \sigma) \to (p + 1, \rho, \sigma) \]