1 Arithmetic

We'll start with a simple expression language. There are two types, integers \( \text{int} \) and booleans \( \text{bool} \). The language consists of literals, a few arithmetic and comparison operators, and a conditional expression.

\[
\begin{align*}
\tau & ::= \text{int} | \text{bool} \\
e & ::= x | \overline{n} | \overline{tt} | \overline{ff} | e + e | e * e | e \le e | \text{if}(e,e,e)
\end{align*}
\]

We define the typing judgment \( \Gamma \vdash e : \tau \) and operational semantics judgments \( \Delta \vdash e \text{ val} \) and \( \Delta \vdash e \mapsto e' \) in Appendix A (N.B. \( \overline{n} \) denotes the term representation of the numeral \( n \) in our simple language). Type contexts \( \Gamma \) and evaluation contexts \( \Delta \) are defined with the following grammar:

\[
\begin{align*}
\Gamma & ::= \cdot | \Gamma, x : \tau \\
\Delta & ::= \cdot
\end{align*}
\]

For the moment, the only evaluation context is the empty context \( \cdot \), but we will change this as we build the language.

Before we add anything new, however, let's check some properties to make sure you're comfortable with rule induction. For the proofs in this assignment, you may condense similar cases in arguments by induction, but be clear and rigorous in your reasoning.

**Task 1** Prove the following inversion lemma:

* (If Inversion) If \( \Gamma \vdash \text{if}(e_1,e_2,e_3) : \tau \), then \( \Gamma \vdash e_1 : \text{bool} \), \( \Gamma \vdash e_2 : \tau \), and \( \Gamma \vdash e_3 : \tau \).

This seems immediate, but really follows from the induction principle for the typing judgment. (Tip: Prove that for all \( \Gamma, e, \tau \) such that \( \Gamma \vdash e : \tau \), if \( e = \text{if}(e_1,e_2,e_3) \) for some \( e_1, e_2, e_3 \), then \( \Gamma \vdash e_1 : \text{bool} \), \( \Gamma \vdash e_2 : \tau \), and \( \Gamma \vdash e_3 : \tau \).)

**Solution:** We show that for all \( \Gamma, e, \tau \) such that \( \Gamma \vdash e : \tau \), if \( e = \text{if}(e_1;e_2;e_3) \), then \( \Gamma \vdash e_1 : \text{bool} \), \( \Gamma \vdash e_2 : \tau \), and \( \Gamma \vdash e_3 : \tau \). We go by induction on the proof of the typing judgment.

- Case (If): In this case, \( e \) has the form \( \text{if}(e'_1;e'_2;e'_3) \) where \( \Gamma \vdash e'_1 : \text{bool} \), \( \Gamma \vdash e'_2 : \tau \), and \( \Gamma \vdash e'_3 : \tau \). Let \( e_1,e_2,e_3 \) be such that \( e = \text{if}(e_1;e_2;e_3) \). Then \( \text{if}(e'_1;e'_2;e'_3) = \text{if}(e_1;e_2;e_3) \), so \( e_1 = e'_1 \), \( e_2 = e'_2 \), and \( e_3 = e'_3 \). The conclusion follows directly.

In all other cases, the assumption \( e = \text{if}(e_1;e_2;e_3) \) is contradictory, because \( e \) is known to have another form. For example, in the (Num) case, we assume \( e \) has the form \( \overline{n} \), and \( \overline{n} = \text{if}(e_1;e_2;e_3) \) leads to a contradiction. Thus the result follows vacuously in these cases.

In general, an inversion lemma is one which recovers the premises of a rule from its conclusion. In the rest of the assignment, you may use these without proof, but be sure to note explicitly when you apply them and check for yourself that they hold.
Task 2 Prove unicity of typing for this language.

(Unicity of Typing) For any $\Gamma$, $e$, $\tau$, $\tau'$ such that $\Gamma \vdash e : \tau$ and $\Gamma \vdash e : \tau'$, we have $\tau = \tau'$.

You may assume that any variable appears at most once in a given context.

Solution: Let $\Gamma$, $e$, $\tau$ be such that $\Gamma \vdash e : \tau$. We show by induction on the proof of the typing judgment that, for any $\tau'$ such that $\Gamma \vdash e : \tau'$, we have $\tau = \tau'$.

- Case (Hyp): In the case $\Gamma \vdash e : \tau$ is derived by a hypothesis $x : \tau \in \Gamma$, inversion on the derivation of $\Gamma \vdash x : \tau'$ gives that $x : \tau' \in \Gamma$. Since any variable appears at most once in a context, we must have $\tau = \tau'$.

- Case (Num): In this case $e = n$ and $\tau = \text{int}$. Thus $\Gamma \vdash n : \tau'$, and by inversion on this judgment we obtain $\tau' = \text{int}$.

The cases (True), (False), (Plus), (Times), and (Leq) follow by the same reasoning, since each rule has a closed type in its conclusion (i.e., one which contains no variables).

- Case (If): In this case $e$ has the form $\text{if}(e_1; e_2; e_3)$ for some $e_1, e_2, e_3$ where $\Gamma \vdash e_1 : \text{bool}$, $\Gamma \vdash e_2 : \tau$, and $\Gamma \vdash e_3 : \tau$. By inversion on the judgment $\Gamma \vdash \text{if}(e_1; e_2; e_3) : \tau'$ we have in particular $\Gamma \vdash e_2 : \tau'$. By induction hypothesis, the theorem holds for the judgment $\Gamma \vdash e_2 : \tau$. Thus the fact that $\Gamma \vdash e_2 : \tau'$ implies $\tau = \tau'$.

2 Functions

On top of the expression language, we now add a syntax for programs $p$, which consist of a series of one-argument function definitions and a final expression computation. (In such a small language, it would be useful to have multi-argument functions, but we will restrict ourselves for the sake of simplicity.) In order to make use of these definitions, we add syntax for function calls to the expression language. The sort of functions consists only of function variables, which we denote here with the letter $u$.

\[
\begin{align*}
f & ::= u \\
e & ::= \cdots \mid f(e) \\
p & ::= \text{fun } u (x : \tau) \{ e \}; p \mid \text{result } e
\end{align*}
\]

We allow calls to previously defined functions to appear in the definitions of later functions as well as in the result expression. As an example, the program

\[
\begin{align*}
\text{fun not } (b : \text{bool}) \{ \text{if}(b, \text{tt}, \text{ff}) \}; \\
\text{fun abs } (x : \text{int}) \{ \text{if}(\text{not}(0 \leq x), -1 + x, x) \}; \\
\text{fun square } (z : \text{int}) \{ z \ast z \}; \\
\text{fun cube } (z : \text{int}) \{ \text{square}(z) \ast z \}; \\
\text{result } \text{cube}(\text{abs}(3)) + \text{cube}(\text{abs}(2))
\end{align*}
\]

should compute to 35.

Task 3 Give a definition in uniform syntax for the language constructs described above. Include the arities (with sorts) of the operators, per Section 1.2 of PFPL.

Solution:

<table>
<thead>
<tr>
<th>concrete</th>
<th>uniform</th>
<th>(arity) sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(e)$</td>
<td>$\text{app}(f; e)$</td>
<td>(fun, exp)exp</td>
</tr>
<tr>
<td>$\text{fun } u (x : \tau) { e }; p$</td>
<td>$\text{fun}[\tau](x; e; u; p)$</td>
<td>(exp, exp, fun, prog)prog</td>
</tr>
<tr>
<td>$\text{result } e$</td>
<td>$\text{res}(e)$</td>
<td>(exp)prog</td>
</tr>
</tbody>
</table>
Before giving the precise operational semantics of this language, we have an ambiguity to settle. Consider the following program.

\[
\begin{align*}
\text{fun } & u (x : \text{int}) \{ x \}; \\
\text{fun } & v (y : \text{int}) \{ u(y) \}; \\
\text{fun } & u (z : \text{int}) \{ z + 1 \}; \\
\text{result } & v(1)
\end{align*}
\]

The result expression \(v(1)\) will be computed in an environment of function definitions. Executing the call \(v(1)\) looks up the definition of \(v\), which then leads to a call to \(u\). But which of the two functions named \(u\) does this call refer to? Depending on how the environment behaves, it may be the first or the second, and so this program might compute either 1 or 2. The first evaluation strategy, in which the \(u\) that \(v\) calls remains the \(u\) available when \(v\) was defined, is called static scoping (sometimes lexical scoping). The second, wherein the call to \(u\) in \(v\) refers to whichever \(u\) was most recently added to the environment, is called dynamic scoping.

**Task 4** We want to respect the identification convention:

Abstract binding trees are always identified up to \(\alpha\)-equivalence. With this in mind, which of the two strategies is correct? Explain.

**Solution:** Static scope is correct. If we use dynamic scope, the program

\[
\begin{align*}
\text{fun } & u (x : \text{int}) \{ x \}; \\
\text{fun } & v (y : \text{int}) \{ u(y) \}; \\
\text{fun } & w (z : \text{int}) \{ z + 1 \}; \\
\text{result } & v(1)
\end{align*}
\]

has a different result than the one shown above, although the two are \(\alpha\)-equivalent, as is clear when we write them in abstract syntax:

\[
\begin{align*}
\text{fun} & (x.x; u.\text{fun}(y.\text{app}(u;y); v.\text{fun}(z.\text{plus}(z;1); u.\text{res}(\text{app}(v;1)))))) \\
\text{fun} & (x.x; u.\text{fun}(y.\text{app}(u;y); v.\text{fun}(z.\text{plus}(z;1); w.\text{res}(\text{app}(v;1))))))
\end{align*}
\]

Now, let’s move on to defining the statics and dynamics of this language precisely. We will need two typing judgments to determine if programs are well-typed. Besides the judgment \(\Gamma \vdash e : \tau\) for typing expressions we have already mentioned, there should be a judgment \(\Gamma \vdash p : \tau\) which states that a program has result of type \(\tau\). We also need to add a new kind of hypothesis to our type contexts:

\[
\Gamma ::= \cdots | u : \tau_1 \Rightarrow \tau_2
\]

The assumption \(u : \tau_1 \Rightarrow \tau_2\) is used to denote that \(u\) is a function variable in scope which takes an argument of type \(\tau_1\) and returns a result of type \(\tau_2\).

**Task 5** Starting with the rules in Appendix A, complete the definition of the typing judgment \(\Gamma \vdash e : \tau\), and define the judgement \(\Gamma \vdash p : \tau\).

**Solution:**

\[
\begin{align*}
\frac{\Gamma \vdash u : \tau_1 \Rightarrow \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash u(e) : \tau_2} \quad \text{(APP)} \\
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{result } e : \tau} \quad \text{(RES)} \\
\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \quad \Gamma, u : \tau_1 \Rightarrow \tau_2 \vdash p : \tau}{\Gamma \vdash (\text{fun } u (x : \tau_1) \{ e \}; p) : \tau} \quad \text{(FUN)}
\end{align*}
\]
Task 6  Give a structural operational semantics for this language by completing the definition of $\Delta \vdash e \mapsto e'$ and defining a judgment $\Delta \vdash p \mapsto p'$. Assume that the value judgment $\Delta \vdash p \text{ val}$ is defined by the single rule

$$\frac{\Delta \vdash e \text{ val}}{\Delta \vdash (\text{result } e) \text{ val}} \quad \text{(Res-V)}$$

In order to track the environment of function definitions, we will add a new kind of assumption to our evaluation context:

$$\Delta ::= \cdot \cdot \cdot | \ u \text{ def } = x.e$$

The assumption $u \text{ def } = x.e$ expresses that the function variable $u$ is currently in scope and that it takes an argument $x$ and computes the expression $e$.

Your definition of the semantics should satisfy the properties of progress and preservation.

(Progress) If $\Gamma \vdash p : \tau$, then either $\Delta \vdash p \text{ val}$ or there exists $p' \text{ such that } \Delta \vdash p \mapsto p'$.

(Preservation) If $\Gamma \vdash p : \tau$ and $\Delta \vdash p \mapsto p'$, then $\Delta \vdash p' : \tau$.

Solution:

$$\frac{\Delta \vdash e \mapsto e'}{\Delta \vdash u(e) \mapsto u(e')} \quad \text{(APP-S)} \quad \frac{\Delta \vdash e \text{ val}}{\Delta \vdash u(e) \mapsto [e/x]e'} \quad \text{(APP-I)}$$

$$\frac{\Delta \vdash (\text{fun } u (x : \tau) \{e\}; p) \mapsto (\text{fun } u (x : \tau) \{e\}; p')}{\Delta \vdash u \text{ def } = x.e \vdash p \mapsto p'} \quad \text{(FUN-S)}$$

$$\frac{\Delta \vdash (\text{fun } u (x : \tau) \{e\}; p) \mapsto p}{\Delta \vdash u \text{ def } = x.e \vdash p \text{ val}} \quad \frac{\Delta \vdash e \mapsto e'}{\Delta \vdash \text{result } e \mapsto \text{result } e'} \quad \text{(Res-S)}$$

Task 7  Prove progress for the rules you have specified. You will want to state and prove an analogous theorem for expressions first. For programs, it may be helpful to prove the following more general theorem:

If $\Delta :: \Gamma$ and $\Gamma \vdash p : \tau$, then either $\Delta \vdash p \text{ val}$ or there exists $p'$ such that $\Delta \vdash p \mapsto p'$.

Here $\Delta :: \Gamma$ is a judgment on contexts defined by the following rules:

$$\frac{}{\cdot ::} \quad \frac{\Delta :: \Gamma \quad \Delta \stackrel{\text{def}}{=} f = x.e}{(\Delta, f \stackrel{\text{def}}{=} x.e) :: (\Gamma, f : \tau_1 \Rightarrow \tau_2)} \quad \text{(ENV-CONS)}$$

This judgment expresses that the environment $\Delta$ and type context $\Gamma$ reference the same function variables, i.e., that they agree on what is in scope. This condition is enough to guarantee a canonical forms lemma, which you will need for your proof:

(Canonical Forms Lemma) Let contexts $\Delta :: \Gamma$ and an expression $e$ be given. Assume that $\Delta \vdash e \text{ val}$. If $\Gamma \vdash e : \text{int}$, then $e = n$ for some $n \in \mathbb{Z}$. If $\Gamma \vdash e : \text{bool}$, then either $e = \text{tt}$ or $e = \text{ff}$.

This lemma enumerates the forms that well-typed values can take (the eponymous canonical forms).

You are not required to prove this lemma.

Solution: We state the progress theorem for expressions:

If $\Delta :: \Gamma$ and $\Gamma \vdash e : \tau$, then either $\Delta \vdash e \text{ val}$ or there exists $e'$ such that $\Delta \vdash e \mapsto e'$.

We go by induction on the typing judgment $\Gamma \vdash e : \tau$.  

4
• Case (HYP): This case follows vacuously, as the judgment $\Delta :: \Gamma$ ensures that $x : \tau \not\in \Gamma$.

• Case (NUM): Then $e$ has the form $\pi$ so $\Delta \vdash e \text{ val}$ by (NUM-V).

The cases for (TRUE) and (FALSE) are analogous to the (NUM) case.

• Case (PLUS): Then $e$ has the form $\text{plus}(e_1; e_2)$ for some $e_1, e_2$ with $\Gamma \vdash e_1 : \text{int}$ and $\Gamma \vdash e_2 : \text{int}$. By induction hypothesis, either there exist $e'_1$ such that $\Delta \vdash e_1 \mapsto e'_1$ or $\Delta \vdash e_1 \text{ val}$. If the former holds, then $\Delta \vdash e \mapsto \text{plus}(e'_1; e_2)$ by (PLUS-S1). In the latter case, by induction hypothesis either there exist $e'_2$ such that $\Delta \vdash e_2 \mapsto e'_2$ or $\Delta \vdash e_2 \text{ val}$. In the former subcase, $\Delta \vdash e \mapsto \text{plus}(e'_1; e'_2)$ by (PLUS-S2). In the latter, we have $e_1 = \pi$ and $e_2 = m$ for some $n, m \in \mathbb{Z}$ by Canonical Forms, and therefore $e \mapsto n + m$ by (PLUS-I).

The (TIMES), (LEQ), and (IF) cases are analogous to the (PLUS) case.

• Case (APP): Then $e = u(e')$ for some $u, e'$ with $\Gamma \vdash u : \tau' \Rightarrow \tau$ and $\Gamma \vdash e' : \tau'$. By induction hypothesis, either $\Delta \vdash e' \text{ val}$ or there is $e''$ with $\Delta \vdash e' \mapsto e''$. In the latter case, $\Delta \vdash e \mapsto u(e'')$ by (APP-S). In the former, we use the fact that $\Gamma \vdash u : \tau' \Rightarrow \tau$ and $\Delta :: \Gamma$, which guarantees (via a simple induction argument) that $u \overset{\text{def}}{=} x.e'' \in \Delta$ for some $x, e''$. By (APP-I), we then have $\Delta \vdash e \mapsto [e'/x]e''$.

Now we prove the general progress theorem for programs. We go by induction on the typing judgment $\Delta \vdash p : \tau$.

• Case (RES): We have $p = \text{result } e$ where $\Gamma \vdash e : \tau$. By the expression progress theorem, either $\Delta \vdash e \text{ val}$ or $\Delta \vdash e \mapsto e'$. In the first case we apply (RES-V), in the second (RES-S).

• Case (FUN): We have $p = \text{fun } (x : \tau_1) \{e\}; p'$ where $\Gamma, x : \tau_1 \vdash e : \tau_2$ and $\Gamma, u : \tau_1 \Rightarrow \tau_2 \vdash p' : \tau$. By the rule (ENV-CONS), we have $(\Delta, u \overset{\text{def}}{=} x.e) :: (\Gamma, u : \tau_1 \Rightarrow \tau_2)$. Therefore we may apply the induction hypothesis to conclude that either $\Delta, u \overset{\text{def}}{=} x.e \vdash p' \text{ val}$ or $\Delta, u \overset{\text{def}}{=} x.e \vdash p' \mapsto p''$ for some $p''$. In the former case we apply (FUN-I), in the latter (FUN-S).

We can make this language a bit more useful by allowing function definitions to refer recursively to the functions they define. We can then write, for example, the factorial function:

\[
\text{fun fact} (x : \text{int}) \{\text{if}(x \leq 0, 1, x * \text{fact}(x + (-1)))\};
\]
\[
\text{result fact}(5)
\]

**Task 8** How does this extension change your answers to Tasks 3 and 4? Give the syntax and rules where different. (You should also make a change to the form of the assumption $u \overset{\text{def}}{=} x.e$.)

**Solution:** For Task 3 the abstract syntax of function declaration changes:

<table>
<thead>
<tr>
<th>Concrete</th>
<th>Abstract</th>
<th>Arity</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun $u$ $(x : \tau)$ $({e}; p)$</td>
<td>fun $\tau(x,u,e;u,p)$</td>
<td>((exp,fun,exp,fun,prog)prog)</td>
</tr>
</tbody>
</table>

For Task 4 the typing rule (FUN) changes:

\[
\frac{\Gamma, x : \tau_1, u : \tau_1 \Rightarrow \tau_2 \vdash e : \tau_2 \quad \Gamma, u : \tau_1 \Rightarrow \tau_2 \vdash p : \tau}{\Gamma \vdash \text{fun } (x : \tau_1) \{e\}; p : \tau} \quad \text{(FUN)}
\]

The judgment $u \overset{\text{def}}{=} x.e$ must be changed to the form:

\[
u \overset{\text{def}}{=} x.u.e
\]
where \( u \) is the recursive reference available in \( e \). For Task 6, the dynamics rules (Fun-I), (Fun-S) and (App-I) change:

\[
\begin{align*}
&\frac{\Gamma, u \triangleq x.u.e : p \mathbin{\downarrow} \text{val}}{\Delta \vdash (\text{fun } u (x : \tau) \{e\}; p) \mapsto p} \quad \text{(Fun-I)} \\
&\frac{\Gamma \vdash e \mathbin{\downarrow} \text{val} \quad \Gamma \vdash u \triangleq x.u.e}{\Gamma \vdash u(e) \mapsto [e/x]e'} \quad \text{(App-I)} \\
&\frac{\Delta, u \triangleq x.u.e : p \mapsto p'}{\Delta \vdash (\text{fun } u (x : \tau) \{e\}; p) \mapsto (\text{fun } u (x : \tau) \{e\}; p')} \quad \text{(Fun-S)}
\end{align*}
\]

A  Base Expression Language

A.1  Statics

\[
\begin{align*}
&\frac{\Gamma, x : \tau \vdash x : \tau}{\Delta \vdash \llbracket x \rrbracket : \tau} \quad \text{(HYP)} \\
&\frac{\text{true} \vdash \text{true}}{\Delta \vdash \text{true} : \text{bool}} \quad \text{(TRUE)} \\
&\frac{\Gamma \vdash e_1 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \quad \text{(PLUS)} \\
&\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash \text{if}(e_1, e_2) : \tau} \quad \text{(IF)}
\end{align*}
\]

A.2  Dynamics

\[
\begin{align*}
&\frac{\Delta \vdash \llbracket \text{val} \rrbracket \quad \Delta \vdash \text{true} : \text{bool}}{\Delta \vdash \text{true} : \text{bool}} \quad \text{(TRUE-V)} \\
&\frac{\Delta \vdash e_1 : \text{val} \quad \Delta \vdash e_2 : \text{val}}{\Delta \vdash e_1 + e_2 : \text{val}} \quad \text{(PLUS-S1)} \\
&\frac{\Delta \vdash m \mapsto n}{\Delta \vdash m + n} \quad \text{(PLUS-I)} \\
&\frac{\Delta \vdash e_1 : \text{val} \quad \Delta \vdash e_2 : \text{val}}{\Delta \vdash e_1 \ast e_2 : \text{val}} \quad \text{(PLUS-S2)} \\
&\frac{\Delta \vdash m \ast n}{\Delta \vdash m \ast n} \quad \text{(PLUS-I)} \\
&\frac{\Delta \vdash e_1 : \text{val}}{\Delta \vdash e_1 \leq e_2} \quad \text{(LEQ-S1)} \\
&\frac{\Delta \vdash e_1 : \text{val} \quad \Delta \vdash e_2 : \text{val}}{\Delta \vdash e_1 \leq e_2} \quad \text{(LEQ-S2)} \\
&\frac{\Delta \vdash m \leq n}{\Delta \vdash m \leq n} \quad \text{(LEQ-I)} \\
&\frac{\Delta \vdash m > n}{\Delta \vdash m \leq n} \quad \text{(LEQ-I2)} \\
&\frac{\Delta \vdash e_1 : \text{val}}{\Delta \vdash \text{if}(e_1, e_2, e_3) \mapsto e_2} \quad \text{(IF-S)} \\
&\frac{\Delta \vdash \text{if}(\text{true}, e_2, e_3) \mapsto \text{if}(e_1, e_2, e_3)}{\Delta \vdash \text{if}(\text{true}, e_2, e_3) \mapsto e_3} \quad \text{(IF-I1)} \\
&\frac{\Delta \vdash \text{if}(\text{false}, e_2, e_3) \mapsto \text{if}(e_1, e_2, e_3)}{\Delta \vdash \text{if}(\text{false}, e_2, e_3) \mapsto e_3} \quad \text{(IF-I2)}
\end{align*}
\]