Computation Exercises 3: Induction

1. Binary trees are a commonly used data structure. Roughly, a binary tree is either a single *leaf node*, or a *branch node* which has two *subtrees*. The set of binary trees can be defined formally by the following grammar:

   $$\texttt{bTree} ::= \texttt{Node} \mid \texttt{Branch}(\texttt{bTree}, \texttt{bTree})$$

   Note the similarity with arithmetic expressions.

   (a) Draw pictures of the following binary trees:

   $$\begin{array}{l}\texttt{Node}\\ \texttt{Branch}(T_1, T_2)\\ \texttt{Branch}(\texttt{Node}, \texttt{Node})\\ \texttt{Branch}(\texttt{Node}, \texttt{Branch}(\texttt{Node}, \texttt{Node}))\end{array}$$

   (b) We define the function `leaves` which take a binary tree as an argument and returns the number of leaf nodes, given by `Node`, in a tree:

   Base Case: $\texttt{leaves}(\texttt{Node}) = 1$
   Inductive Case: $\texttt{leaves}(\texttt{Branch}(T_1, T_2)) = \texttt{leaves}(T_1) + \texttt{leaves}(T_2)$.

   We now define another function, `branches`, which counts the number of $\texttt{Branch}(\_, \_)$ nodes in a tree:

   Base Case: $\texttt{branches}(\texttt{Node}) = 0$
   Inductive Case:

   $$\texttt{branches}(\texttt{Branch}(T_1, T_2)) = \texttt{branches}(T_1) + \texttt{branches}(T_2) + 1.$$

   Prove by induction on the structure of trees that, for any tree T,

   $$\texttt{leaves}(T) = \texttt{branches}(T) + 1.$$

2. (**Totality of $\Downarrow$**) Recall the **big-step** operational semantics for simple expressions E. Prove by structural induction on the structure of expressions that, for every E, there is some number n such that $E \Downarrow n$.

3. Recall the function `den` from the lectures. For every simple expression E and number n, prove that

   $$\texttt{den}(E) = n \text{ if and only if } E \Downarrow n$$

4. (**$\rightarrow$-normal forms**) Recall the **small-step** operational semantics for simple expressions. (Consider only the operations $+$ and $\times$.) Prove, by induction on the structure of simple expressions, that for every expression E, either $E = n$ for some number n, or $E \rightarrow E'$ for some expression E'.

5. (**Normalisation for $\rightarrow$**) Recall the **small-step** operational semantics for simple expressions.
   (a) By induction on the structure of simple expressions, define a function
   $$\mathit{ops} : \mathit{SimpleExp} \rightarrow \mathbb{N}$$
   that gives the number of operators (instances of $+$ and $\times$) in an expression. (You need only consider the operators $+$ and $\times$.)
   (b) By induction on the structure of simple expressions, prove that for all simple expressions E, E' with $E \rightarrow E'$, $\mathit{ops}(E) > \mathit{ops}(E')$.
   (c) Hence or otherwise, prove that $\rightarrow$ is normalising.

6. For any simple expression E, prove by induction on the structure of expressions that:
   $$E \Downarrow n \text{ if and only if } E \rightarrow^* n$$

7. (**Determinacy for *While***) Prove, by induction on the structure of commands, that the small-step semantics for *While* is deterministic: that is, for any configurations $\langle C, s\rangle$, $\langle C_1, s_1\rangle$, $\langle C_2, s_2\rangle$, if $\langle C, s\rangle \rightarrow_c \langle C_1, s_1\rangle$ and $\langle C, s\rangle \rightarrow_c \langle C_2, s_2\rangle$ then $\langle C_1, s_1\rangle = \langle C_2, s_2\rangle$. You may assume that $\rightarrow_e$ and $\rightarrow_b$ are deterministic.

The following questions are more difficult than those in previous sheets, and some of them should be quite challenging. It might be useful if you attempt them before next week's tutorial, when you can ask about any problems you encounter.

8. Consider the following **big step** semantics for program expressions:

$$\textsc{Num}\ \frac{}{\langle n, s\rangle \Downarrow \langle n, s\rangle} \qquad \textsc{Var}\ \frac{}{\langle x, s\rangle \Downarrow \langle s(x), s\rangle}$$

$$\textsc{Plus}\ \frac{\langle E_1, s\rangle \Downarrow \langle n_1, s\rangle \qquad \langle E_2, s\rangle \Downarrow \langle n_2, s\rangle}{\langle E_1 + E_2, s\rangle \Downarrow \langle n_3, s\rangle}\ n_3 = n_1 \underline{+} n_2$$

   (a) Complete these semantics by adding a rule for multiplication.
   (b) Let $f : \mathit{Exp} \rightarrow \mathit{Exp}$ be the expression transformation function defined by induction as:
   $$\begin{aligned} f(x) &= x \\ f(n) &= n \\ f(E_1 + E_2) &= \begin{cases} 2 \times f(E_1) & \text{if } f(E_1) = f(E_2) \\ f(E_1) + f(E_2) & \text{otherwise} \end{cases} \\ f(E_1 \times E_2) &= f(E_1) \times f(E_2) \end{aligned}$$
   A compiler might use a transformation such as this to optimise a program, since it simplifies expressions. It is important that optimisations preserve the behaviour of programs: any behaviour of $f(E)$ must also be a behaviour of E.
   Prove by induction on the structure of expressions, that for all expressions E, numbers n and states s,
   $$\langle f(E), s\rangle \Downarrow \langle n, s\rangle \implies \langle E, s\rangle \Downarrow \langle n, s\rangle$$

9. Suppose that S is a set ranged over by $s, s_1, s_2, \dots$, and that $\rightarrowtail \subseteq S \times S$ is a binary relation over this set. (For example, S could be the set of program configurations and $\rightarrowtail$ the small-step transition relation.) One way of defining the reflexive transitive closure, $\rightarrowtail^*$, of $\rightarrowtail$ is by the following two derivation rules:

$$\text{REFL}\ \frac{}{s \rightarrowtail^* s} \qquad \text{STEP}\ \frac{s_1 \rightarrowtail^* s_2 \qquad s_2 \rightarrowtail s_3}{s_1 \rightarrowtail^* s_3}$$

   (a) Show, by induction on the structure of derivations, that $\rightarrowtail^*$ is transitive. That is, show: for all $s_1, s_2, s_3 \in S$, if $s_1 \rightarrowtail^* s_2$ and $s_2 \rightarrowtail^* s_3$ then $s_1 \rightarrowtail^* s_3$.

   (Hint: you should do induction on the derivation of $s_2 \rightarrowtail^* s_3$; if you try to do the induction on the derivation of $s_1 \rightarrowtail^* s_2$, you will find it difficult to make progress.)

   (b) Show that the reflexive transitive closure operator is idempotent. That is, show: for all $s_1, s_2 \in S$, $s_1 \rightarrowtail^* s_2$ if and only if $s_1 (\rightarrowtail^*)^* s_2$.

   Note, the derivation rules as they apply to $(\rightarrowtail^*)^*$ are:

$$\text{REFL}\ \frac{}{s\ (\rightarrowtail^*)^*\ s} \qquad \text{STEP}\ \frac{s_1\ (\rightarrowtail^*)^*\ s_2 \qquad s_2 \rightarrowtail^* s_3}{s_1\ (\rightarrowtail^*)^*\ s_3}$$

Remember that in Exercises 1, Question 8 we saw a variation of the small-step operational semantics for *SimpleExpr* that was confluent but not deterministic. In the answer sheet, we said that a good way to show confluence is to show *strong confluence*. We can show that a strongly confluent relation is also confluent. By answering the next two questions, you will show both of these results.

10. (**Strong Confluence implies Confluence**) Suppose that $\leadsto$ is a strongly confluent rewrite relation (perhaps similar to our small-step operational semantics for *SimpleExpr*). That is, for all t, t_1, t_2 such that $t \leadsto t_1$ and $t \leadsto t_2$, either $t_2 \leadsto^* t_1$ or there exists some t' such that $t_1 \leadsto t'$ and $t_2 \leadsto^* t'$. We can represent this by the following diagram:

$$\begin{tikzcd}
 & t \arrow[dl, "1"'] \arrow[dr, "1"] & \\
t_1 \arrow[dr, dashed, "{\{0,1\}}"'] & & t_2 \arrow[dl, dashed, "*"] \\
 & t' &
\end{tikzcd}$$

Prove that $\leadsto$ is confluent. That is, for all t, t_1, t_2 such that $t \leadsto^* t_1$ and $t \leadsto^* t_2$ there exists some t' such that $t_1 \leadsto^* t'$ and $t_2 \leadsto^* t'$. We can represent this by the following diagram:

$$\begin{tikzcd}
 & t \arrow[dl, "*"'] \arrow[dr, "*"] & \\
t_1 \arrow[dr, dashed, "*"'] & & t_2 \arrow[dl, dashed, "*"] \\
 & t' &
\end{tikzcd}$$

It will probably be easiest if you do the proof in the following two stages:

(a) Prove, by induction on n, $\forall n.\, P_1(n)$, where

$$P_1(n) \equiv \forall t, t_1, t_2.\, t \rightsquigarrow^n t_1 \wedge t \rightsquigarrow t_2 \implies t_2 \rightsquigarrow^* t_1 \vee \exists t'.\, t_1 \rightsquigarrow t' \wedge t_2 \rightsquigarrow^* t'$$

You will use strong confluence of $\rightsquigarrow$ to establish the inductive step. (As a diagram, the inductive step will look like:

$$\begin{tikzcd}
 & & t \arrow[dl, "n"'] \arrow[dr, "1"] & \\
 & t_1' \arrow[dl, "1"'] \arrow[dr, dashed, "\{0,1\}"'] & & t_2 \arrow[dl, dashed, "*"] \\
t_1 \arrow[dr, dashed, "\{0,1\}"'] & & t'' \arrow[dl, dashed, "*"] & \\
 & t' & &
\end{tikzcd}$$

The upper-right part of the diagram is established by the inductive hypothesis. The lower-left part is established by strong confluence.)

(b) Prove, by induction on m, $\forall m.\, P_2(m)$, where

$$P_2(m) \equiv \forall t, t_1, t_2.\, t \rightsquigarrow^* t_1 \wedge t \rightsquigarrow^m t_2 \implies \exists t'.\, t_1 \rightsquigarrow^* t' \wedge t_2 \rightsquigarrow^* t'$$

You will use 10a to establish the inductive step. (As a diagram, the inductive step will look like:

$$\begin{tikzcd}
 & t \arrow[dl, "*"'] \arrow[dr, "m"] & & \\
t_1 \arrow[dr, dashed, "*"'] & & t_2' \arrow[dl, dashed, "*"] \arrow[dr, "1"] & \\
 & t'' \arrow[dr, dashed, "\{0,1\}"'] & & t_2 \arrow[dl, dashed, "*"] \\
 & & t' &
\end{tikzcd}$$

The upper-left part of the diagram is established by the inductive hypothesis. The lower-right part is established using 10a.)

11. Recall the modified small-step semantics for *SimpleExp* that we saw in Question 8 of Exercise sheet 1, which had the following three rules:

$$(\text{S-LEFT})\ \frac{E_1 \to E_1'}{(E_1 + E_2) \to (E_1' + E_2)} \qquad (\text{S-RIGHT'})\ \frac{E_2 \to E_2'}{(E_1 + E_2) \to (E_1 + E_2')}$$

$$(\text{S-ADD})\ \frac{}{(n_1 + n_2) \to n_3}\ n_3 = n_1 \underline{+} n_2$$

Prove that $\to$ is strongly confluent. That is, for all E, E_1, E_2 such that $E \to E_1$ and $E \to E_2$, either $E_2 \to^* E_1$ or there exists some E' such that $E_1 \to E'$ and $E_2 \to^* E'$.

To establish this, it is sufficient to prove

$$E \to E_1 \wedge E \to E_2 \implies E_1 = E_2 \vee \exists E'.\, E_1 \to E' \wedge E_2 \to E'$$

This may be done by induction on the structure of the derivation of $E \to E_1$.

12. Prove that the ; operator is associative by induction on the length of computation sequences. That is, prove that

$$\langle C_1; (C_2; C_3), s\rangle \to^* \langle \texttt{skip}, s'\rangle \text{ if and only if } \langle (C_1; C_2); C_3, s\rangle \to^* \langle \texttt{skip}, s'\rangle.$$

(A good approach is to prove that, for all n,

$$\langle C_1; (C_2; C_3), s\rangle \to^n \langle \texttt{skip}, s'\rangle \quad \text{if and only if} \quad \langle (C_1; C_2); C_3, s\rangle \to^n \langle \texttt{skip}, s'\rangle.$$

by induction on n.)

13. [Historical aside:] In this course, we look at lots of models of computation: While, register machines, Turing machines and the lambda calculus. One thing these models have in common, which we will look at later, is Turing completeness: anything that is computable in one model is computable in the others. Among the oldest Turing-complete models of computation are combinator calculi, developed by Schönfinkel and Curry (after whom the Haskell language is named) in the 1920s. (Lambda calculus and Turing machines were introduced (by Church and Turing respectively) in the 1930s. The notion of Turing completeness didn't exist when combinator calculi were developed.)

The best-known combinator calculus is the *SKI* combinator calculus. The abstract syntax of *terms* of *SKI*, which are ranged over by $t, t', t_1, x, y, z \ldots$, is defined by the following grammar:

$$t ::= \mathbf{S} \mid \mathbf{K} \mid \mathbf{I} \mid t_1 t_2$$

Thus, the *combinators* **S**, **K** and **I** are terms, and for any two terms x and y, the *application* of x to y, (xy) is also a term. Of course $((xy)z)$ and $(x(yz))$ are different terms. Application is considered to associate to the left, so xyz represents $((xy)z)$.

The *reduction* relation $\to$ on terms is like a small-step operational semantics. It is given by the following derivation rules:

$$\textsc{ap-l}\ \frac{x \to x'}{xy \to x'y} \qquad \textsc{ap-r}\ \frac{y \to y'}{xy \to xy'}$$

$$\textsc{red-I}\ \frac{}{\mathbf{I}x \to x} \qquad \textsc{red-K}\ \frac{}{\mathbf{K}xy \to x} \qquad \textsc{red-S}\ \frac{}{\mathbf{S}xyz \to xz(yz)}$$

You can think of combinators as functions. **I** returns its argument; **K** takes two arguments and returns the first; **S** takes three arguments and applies the first to the third and the application of the second to the third.

Reductions can happen at any level in the term. For instance, the term **I**(**KIS**) can reduce to both **KIS** (by red-I) and **II** (by ap-r with red-K); both of these terms reduce to **I** (by red-K and red-I respectively).

(a) Prove that, for all x, x', y, y', if $x \to^* x'$ and $y \to^* y'$ then $xy \to x'y'$. (If you do the proof by induction on (the number of steps in) the derivation of $x \to^* x'$, you will need to do induction on the derivation of $y \to^* y'$ to prove the base case.)

(b) Prove that $\to$ is *locally confluent*. That is, for all t, t_1, t_2, if $t \to t_1$ and $t \to t_2$ then there exists t' such that $t_1 \to^* t'$ and $t_2 \to^* t'$. (A good approach to this is by induction on the derivation of $t \to t_1$; to prove each case, you then consider the possible cases for the derivation of $t \to t_2$.)