Exercise 2.1  Substitution Lemma

A syntactic substitution replaces a variable by an expression.
Define a function $\text{subst} :: \text{vname} \Rightarrow \text{aexp} \Rightarrow \text{aexp} \Rightarrow \text{aexp}$ that performs a syntactic substitution, i.e., $\text{subst} x a' a$ shall be the expression $a$ where every occurrence of variable $x$ has been replaced by expression $a'$.

Instead of syntactically replacing a variable $x$ by an expression $a'$, we can also change the state $s$ by replacing the value of $x$ by the value of $a'$ under $s$. This is called semantic substitution.

The substitution lemma states that semantic and syntactic substitution are compatible. Prove the substitution lemma:

$\text{lemma subst lemma: } \text{aval} (\text{subst} x a' a) s = \text{aval} a \left( s(x:=\text{aval} a' s) \right)$

Note: The expression $s(x:=v)$ updates a function at point $x$. It is defined as:

$f(a := b) = (\lambda x. \text{if } x = a \text{ then } b \text{ else } f x)$

Compositionality means that one can replace equal expressions by equal expressions. Use the substitution lemma to prove compositionality of arithmetic expressions:

$\text{lemma comp: } \text{aval} a1 s = \text{aval} a2 s \implies \text{aval} (\text{subst} x a1 a) s = \text{aval} (\text{subst} x a2 a) s$"
Define the datatype of extended arithmetic expressions. Hint: If you do not want to hide the standard constructor names from IMP, add a tick (′) to them, e.g., \( V' \ x \).

The semantics of extended arithmetic expressions has the type \( \text{aval}': \text{aexp}' \Rightarrow \text{state} \Rightarrow (\text{val} \times \text{state}) \ \text{option} \), i.e., it takes an expression and a state, and returns a value and a new state, or an error value. Define the function \( \text{aval}' \).

(Hint: To make things easier, we recommend an incremental approach to this exercise: First define arithmetic expressions with incrementing, but without division. The function \( \text{aval}' \) for this intermediate language should have type \( \text{aexp}' \Rightarrow \text{state} \Rightarrow \text{val} \times \text{state} \). After completing the entire exercise with this version, then modify your definitions to add division and exceptions.)

Test your function for some terms. Is the output as expected? Note: \(<>\) is an abbreviation for the state that assigns every variable to zero:

\(<> \equiv \lambda x. 0\)

\begin{align*}
\text{value} & \quad \text{“aval}' (\text{Div}' (V' ''x'') (V' ''x'')) <>” \\
\text{value} & \quad \text{“aval}' (\text{Div}' (\text{PI}' ''x'') (V' ''x'')) <"x":=1>” \\
\text{value} & \quad \text{“aval}' (\text{Plus}' (\text{PI}' ''x'') (V' ''x'')) <>” \\
\text{value} & \quad \text{“aval}' (\text{Plus}' (\text{PI}' ''x'') (\text{PI}' ''x'')) (\text{PI}' ''x'')) <>”
\end{align*}

Is the plus-operation still commutative? Prove or disprove!

Show that the valuation of a variable cannot decrease during evaluation of an expression:

\text{lemma} \text{aval}',inc: “aval}' a s = \text{Some} (v,s') \Rightarrow s x \leq s' x”

Hint: If \text{auto} on its own leaves you with an if in the assumptions or with a case-statement, you should modify it like this: (auto split: split if asm option.splits).

\section*{Homework 2.1 Constant Multiplication}

\textit{Submission until Tuesday, October 29, 10:00am.}

Write a function \text{multn} that takes a natural number \( n::\text{nat} \) and an arithmetic expressions \( a::\text{aexp} \), and returns an arithmetic expression \( b \) such that for all states \( s::\text{state} \):

\( \text{aval} (\text{multn} n a) s = \text{int} n \ast \text{aval} a s \).

Prove that your function is correct.

\begin{align*}
\text{fun} & \quad \text{multn} :: \ “\text{nat} \Rightarrow \text{aexp} \Rightarrow \text{aexp}” \ \text{where} \\
\text{lemma} & \quad \text{“aval} (\text{multn} n a) s = \text{int} n \ast \text{aval} a s”
\end{align*}
Homework 2.2  Tail-Recursive Counting

Submission until Tuesday, October 29, 10:00am.

The list-reversing function $itrev$ is an example of a tail-recursive function: Note that the right-hand side of the second equation for $itrev$ is simply an application of $itrev$ to some arguments.

\[
\text{fun } itrev :: \text{"'a list } \Rightarrow \text{'a list" where}
\]

$itrev \; [] \; ys = ys$

$itrev \; (x\#xs) \; ys = itrev \; xs \; (x\#ys)$

In this homework problem you will define in Isabelle a tail-recursive version of $\text{count}$ (which counts the number of occurrences of a particular element in a list), using an auxiliary argument. More precisely, you should define a function $\text{count\_tr :: 'a list } \Rightarrow \text{'a } \Rightarrow \text{nat } \Rightarrow \text{nat}$ such that $\forall \text{xs y. count\_tr xs y 0 = count xs y}$. Like $itrev$, your definition should be tail-recursive: in the recursive case the right-hand side should only consist of if-expressions, case-distinctions and recursive applications of $\text{count\_tr}$.

First you need to define the function:

\[
\text{primrec } count\_tr :: \text{"'a list } \Rightarrow \text{'a } \Rightarrow \text{nat } \Rightarrow \text{nat" where}
\]

Then you need to prove that $\text{count\_tr}$ is correct. Here, $\text{count}$ is the function from exercise 1.2, you can copy it from the sample solution.

\[
\text{lemma } \text{"count\_tr xs y 0 = count xs y"}
\]

Hint: In order to prove the above lemma, you may first need to prove a more general fact about $\text{count\_tr}$ (employing an arbitrary argument $n$ instead of 0), of which the above lemma is a particular case.

Homework 2.3  Disjunctive Normal Form

Submission until Tuesday, October 29, 10:00am.

Note: This is a “bonus” assignment worth five additional points, making the maximum possible score for all homework on this sheet 15 out of 10 points.

Warning: This assignment is quite hard. Also partial solutions will be graded!

In this assignment, you shall write a function that converts a boolean expression over variables, conjunction, disjunction, and negation to disjunctive normal form, and prove its correctness. A template for this homework is available on the lecture’s homepage.

We start by defining our version of boolean expressions:

\[
\text{datatype } bexp = \text{Not } bexp | \text{And } bexp bexp | \text{Or } bexp bexp | \text{Var vname}
\]

\[
\text{fun } beval :: \text{"bexp } \Rightarrow \text{state } \Rightarrow \text{bool" — Definition in template}
\]

Next, we define functions that check whether a boolean expression is in DNF or NNF.
fun is_dnf :: "bexp ⇒ bool" — Definition in template
fun is_nnf :: "bexp ⇒ bool" — Definition in template

An approach to convert a boolean expression to DNF is to first convert it to NNF
(negation normal form), and then apply the distributivity laws to get the DNF. Thus,
start with defining a function that converts any boolean expression to NNF. This can
be done by "pushing in" negations, and eliminating double negations.

fun to_nnf :: "bexp ⇒ bexp" where

Prove that your function is correct. Hint: use the induction rule generated by the function
package, the syntax is: induction b rule: to_nnf.induct
lemma [simp]: "is_nnf (to_nnf b)"
lemma [simp]: "bval (to_nnf b) s = bval b s"

The basic idea of converting an NNF to DNF is to first convert the operands of a
conjunction, and then apply the distributivity law to get a disjunction of conjunctions.
The function merge \( (a_1 \lor \ldots \lor a_n) (b_1 \lor \ldots \lor b_m) \) shall return a term of the form
\( a_1 \land b_1 \lor a_1 \land b_2 \lor \ldots \lor a_n \land b_m \) that is equivalent to
\( (a_1 \lor \ldots \lor a_n) \land (b_1 \lor \ldots \lor b_m) \).

fun merge :: "bexp ⇒ bexp ⇒ bexp" where

Show that merge preserves the semantics and indeed yields a DNF, if its operands are
already in DNF. Hint: For functions over multiple arguments, the syntax for induction
is induction a b rule: merge.induct
lemma [simp]: "bval (merge a b) s ←→ bval (And a b) s"
lemma [simp]: "is_dnf a \implies is_dnf b \implies is_dnf (merge a b)"

Next, use merge to write a function that converts an NNF to a DNF. The idea is to
first convert the operands of a compound expression, and then compute the overall DNF
(using merge in the And-case)
fun nnf_to_dnf :: "bexp ⇒ bexp" where

Prove the correctness of your function:
lemma [simp]: "bval (nnf_to_dnf b) s = bval b s"
lemma [simp]: "is_nnf b \implies is_dnf (nnf_to_dnf b)"

Finally, combine the two functions to_nnf and nnf_to_dnf, to get a function that converts
any boolean expression to DNF:
definition convert_to_dnf :: "bexp ⇒ bexp"

theorem [simp]: "bval (convert_to_dnf b) s = bval b s"
theorem [simp]: "is_dnf (convert_to_dnf b)"