This document collects together the exercises in the book *Concrete Semantics* in extended form. The exercises are described in more detail and with additional hints. Corresponding templates for solutions are available from the [home page of the book](#) in the form of theory files. Exercises that do not require Isabelle are omitted.
Chapter 2

Exercise 2.1. Use the value command to evaluate the following expressions:

"1 + (2:nat)" "1 + (2:int)" "1 - (2:nat)" "1 - (2:int)" "[a, b] @ [c, d]"

Exercise 2.2. Recall the definition of our own addition function on nat:

fun add :: "nat ⇒ nat ⇒ nat" where
"add 0 n = n" |
"add (Suc m) n = Suc(add m n)"

Prove that add is associative and commutative. You will need additional lemmas.

lemma add_assoc: "add (add m n) p = add m (add n p)"
lemma add_comm: "add m n = add n m"

Define a recursive function

fun double :: "nat ⇒ nat"

and prove that

lemma double_add: "double m = add m m"

Exercise 2.3. Define a function that counts the number of occurrences of an element in a list:

fun count :: "'a list ⇒ 'a ⇒ nat"

Test your definition of count on some examples. Prove the following inequality:

theorem "count xs x ⩽ length xs"

Exercise 2.4. Define a function snoc that appends an element to the end of a list. Do not use the existing append operator @ for lists.

fun snoc :: "'a list ⇒ 'a ⇒ 'a list"

Convince yourself on some test cases that your definition of snoc behaves as expected. With the help of snoc define a recursive function reverse that reverses a list. Do not use the predefined function rev.

fun reverse :: "'a list ⇒ 'a list"

Prove the following theorem. You will need an additional lemma.

theorem "reverse (reverse xs) = xs"

Exercise 2.5. The aim of this exercise is to prove the summation formula

\[ \sum_{i=0}^{n} i = \frac{n(n + 1)}{2} \]

Define a recursive function sum n = 0 + ... + n:
fun sum :: "nat ⇒ nat"

Now prove the summation formula by induction on \( n \). First, write a clear but informal proof by hand following the examples in the main text. Then prove the same property in Isabelle:

**Lemma** "\( \text{sum } n = n \ast \lfloor \frac{n+1}{2} \rfloor \)"

**Exercise 2.6.** Starting from the type ‘a tree defined in the text, define a function that collects all values in a tree in a list, in any order, without removing duplicates.

**Datatype** ‘a tree = Tip | Node "'a tree" 'a "'a tree"

**Function** contents :: "'a tree ⇒ 'a list"

Then define a function that sums up all values in a tree of natural numbers

**Function** treesum :: "nat tree ⇒ nat"

and prove

**Lemma** "treesum t = listsum(contents t)"

**Exercise 2.7.** Define a new type ‘a tree2 of binary trees where values are also stored in the leaves of the tree. Also reformulate the mirror function accordingly. Define two functions

**Function** pre_order :: "'a tree2 ⇒ 'a list"

**Function** post_order :: "'a tree2 ⇒ 'a list"

that traverse a tree and collect all stored values in the respective order in a list. Prove

**Lemma** "pre_order (mirror t) = rev (post_order t)"

**Exercise 2.8.** Define a recursive function

**Function** intersperse :: "'a ⇒ 'a list ⇒ 'a list"

such that intersperse a [\( x_1, \ldots, x_n \)] = [\( x_1, a, x_2, a, \ldots, a, x_n \)]. Prove

**Lemma** "map f (intersperse a xs) = intersperse (f a) (map f xs)"

**Exercise 2.9.** Write a tail-recursive variant of the add function on nat:

**Function** itadd :: "nat ⇒ nat ⇒ nat"

Tail-recursive means that in the recursive case, itadd needs to call itself directly:

itadd (Suc m) n = itadd n ...

Prove

**Lemma** "itadd m n = add m n"
Exercise 2.10. Define a datatype `tree0` of binary tree skeletons which do not store any information, neither in the inner nodes nor in the leaves. Define a function that counts the number of all nodes (inner nodes and leaves) in such a tree:

```haskell
fun nodes :: "tree0 ⇒ nat"
```

Consider the following recursive function:

```haskell
fun explode :: "nat ⇒ tree0 ⇒ tree0" where
  "explode 0 t = t" |
  "explode (Suc n) t = explode n (Node t t)"
```

Experiment how `explode` influences the size of a binary tree and find an equation expressing the size of a tree after exploding it `(nodes (explode n t))` as a function of `nodes t` and `n`. Prove your equation. You may use the usual arithmetic operations including the exponentiation operator `^`. For example, `2 ^ 2 = 4`.

Hint: simplifying with the list of theorems `algebra_simps` takes care of common algebraic properties of the arithmetic operators.

Exercise 2.11. Define arithmetic expressions in one variable over integers (type `int`) as a data type:

```haskell
datatype exp = Var | Const int | Add exp exp | Mult exp exp
```

Define a function `eval` that evaluates an expression at some value:

```haskell
fun eval :: "exp ⇒ int ⇒ int"
```

For example, `eval (Add (Mult (Const 2) Var) (Const 3)) i = 2 * i + 3`.

A polynomial can be represented as a list of coefficients, starting with the constant. For example, `[4, 2, -1, 3]` represents the polynomial `4 + 2x - x^2 + 3x^3`. Define a function `evalp` that evaluates a polynomial at a given value:

```haskell
fun evalp :: "int list ⇒ int ⇒ int"
```

Define a function `coeffs` that transforms an expression into a polynomial. This will require auxiliary functions.

```haskell
fun coeffs :: "exp ⇒ int list"
```

Prove that `coeffs` preserves the value of the expression:

```haskell
theorem evalp_coeffs: "evalp (coeffs e) x = eval e x"
```

Hint: consider the hint in Exercise 2.10.
Chapter 3

Exercise 3.1. To show that `asimp_const` really folds all subexpressions of the form `Plus (N i) (N j)`, define a function

```haskell
fun optimal :: "aexp ⇒ bool"
```

that checks that its argument does not contain a subexpression of the form `Plus (N i) (N j)`. Then prove that the result of `asimp_const` is optimal:

```haskell
lemma "optimal (asimp_const a)"
```

This proof needs the same `split:` directive as the correctness proof of `asimp_const`. This increases the chance of nontermination of the simplifier. Therefore `optimal` should be defined purely by pattern matching on the left-hand side, without `case` expressions on the right-hand side.

Exercise 3.2. In this exercise we verify constant folding for `aexp` where we sum up all constants, even if they are not next to each other. For example, `Plus (N 1) (Plus (V x) (N 2))` becomes `Plus (V x) (N 3)`. This goes beyond `asimp`.

Below we follow a particular solution strategy but there are many others.

First, define a function `sumN` that returns the sum of all constants in an expression and a function `zeroN` that replaces all constants in an expression by zeroes (they will be optimized away later):

```haskell
fun sumN :: "aexp ⇒ int"
fun zeroN :: "aexp ⇒ aexp"
```

Next, define a function `sepN` that produces an arithmetic expression that adds the results of `sumN` and `zeroN`. Prove that `sepN` preserves the value of an expression.

```haskell
definition sepN :: "aexp ⇒ aexp"
lemma aval_sepN: "aval (sepN t) s = aval t s"
```

Finally, define a function `full_asimp` that uses `asimp` to eliminate the zeroes left over by `sepN`. Prove that it preserves the value of an arithmetic expression.

```haskell
definition full_asimp :: "aexp ⇒ aexp"
lemma aval_full_asimp: "aval (full_asimp t) s = aval t s"
```

Exercise 3.3. Substitution is the process of replacing a variable by an expression in an expression. Define a substitution function

```haskell
fun subst :: "vname ⇒ aexp ⇒ aexp ⇒ aexp"
```

such that `subst x a e` is the result of replacing every occurrence of variable `x` by `a` in `e`. For example:

```plaintext
subst "x" (N 3) (Plus (V "x") (V "y")) = Plus (N 3) (V "y")
```

Prove the so-called substitution lemma that says that we can either substitute first and evaluate afterwards or evaluate with an updated state:
\textbf{lemma} subst_lemma: "$\text{aval} \ (\text{subst} \ x \ a \ e) \ s = \text{aval} \ e \ (s(x := \text{aval} \ a \ s))$"

As a consequence prove that we can substitute equal expressions by equal expressions and obtain the same result under evaluation:

\textbf{lemma} "$\text{aval} \ a_1 \ s = \text{aval} \ a_2 \ s$ \\
$\implies \text{aval} \ (\text{subst} \ x \ a_1 \ e) \ s = \text{aval} \ (\text{subst} \ x \ a_2 \ e) \ s$"

Exercise 3.4. Take a copy of theory $\text{AExp}$ and modify it as follows. Extend type $\text{aexp}$ with a binary constructor $\text{Times}$ that represents multiplication. Modify the definition of the functions $\text{aval}$ and $\text{asimp}$ accordingly. You can remove $\text{asimp\_const}$. Function $\text{asimp}$ should eliminate 0 and 1 from multiplications as well as evaluate constant subterms. Update all proofs concerned.

Exercise 3.5. Define a datatype $\text{aexp2}$ of extended arithmetic expressions that has, in addition to the constructors of $\text{aexp}$, a constructor for modelling a C-like post-increment operation $x++$, where $x$ must be a variable. Define an evaluation function $\text{aval2} :: \text{aexp2} \Rightarrow \text{state} \Rightarrow \text{val} \times \text{state}$ that returns both the value of the expression and the new state. The latter is required because post-increment changes the state.

Extend $\text{aexp2}$ and $\text{aval2}$ with a division operation. Model partiality of division by changing the return type of $\text{aval2}$ to $(\text{val} \times \text{state})$ \text{option}. In case of division by 0 let $\text{aval2}$ return $\text{None}$. Division on int is the infix $\text{div}$.

Exercise 3.6. The following type adds a $\text{LET}$ construct to arithmetic expressions:

\text{datatype} \ 	ext{lexp} = \text{Nl} \ \text{int} \mid \text{Vl} \ \text{vname} \mid \text{Plusl} \ \text{lexp} \ \text{lexp} \mid \text{LET} \ \text{vname} \ \text{lexp} \ \text{lexp}

The $\text{LET}$ constructor introduces a local variable: the value of $\text{LET} \ x \ e_1 \ e_2$ is the value of $e_2$ in the state where $x$ is bound to the value of $e_1$ in the original state. Define a function $\text{lval} :: \text{lexp} \Rightarrow \text{state} \Rightarrow \text{int}$ that evaluates $\text{lexp}$ expressions. Remember $s(x := i)$.

Define a conversion $\text{inline} :: \text{lexp} \Rightarrow \text{aexp}$. The expression $\text{LET} \ x \ e_1 \ e_2$ is inlined by substituting the converted form of $e_1$ for $x$ in the converted form of $e_2$. See Exercise 3.3 for more on substitution. Prove that $\text{inline}$ is correct w.r.t. evaluation.

Exercise 3.7. Show that equality and less-or-equal tests on $\text{aexp}$ are definable

\text{definition} \ Le :: "$\text{aexp} \Rightarrow \text{aexp} \Rightarrow \text{bexp}$"
\text{definition} \ Eq :: "$\text{aexp} \Rightarrow \text{aexp} \Rightarrow \text{bexp}$"

and prove that they do what they are supposed to:

\textbf{lemma} bval_Lc $\text{bval} \ (\text{Le} \ a_1 \ a_2) \ s = (\text{aval} \ a_1 \ s \leq \text{aval} \ a_2 \ s)"
\textbf{lemma} bval_Eq $\text{bval} \ (\text{Eq} \ a_1 \ a_2) \ s = (\text{aval} \ a_1 \ s = \text{aval} \ a_2 \ s)"

Exercise 3.8. Consider an alternative type of boolean expressions featuring a conditional:

\textbf{definition} \ Le :: "$\text{aexp} \Rightarrow \text{aexp} \Rightarrow \text{bexp}$"
\text{definition} \ Eq :: "$\text{aexp} \Rightarrow \text{aexp} \Rightarrow \text{bexp}$"

and prove that they do what they are supposed to:

\textbf{lemma} bval_Lc $\text{bval} \ (\text{Le} \ a_1 \ a_2) \ s = (\text{aval} \ a_1 \ s \leq \text{aval} \ a_2 \ s)"
\textbf{lemma} bval_Eq $\text{bval} \ (\text{Eq} \ a_1 \ a_2) \ s = (\text{aval} \ a_1 \ s = \text{aval} \ a_2 \ s)"

Exercise 3.8. Consider an alternative type of boolean expressions featuring a conditional:
datatype ifexp = Bc2 bool | If ifexp ifexp ifexp | Less2 aexp aexp

First define an evaluation function analogously to bval:

fun ifval :: "ifexp ⇒ state ⇒ bool"

Then define two translation functions

fun b2ifexp :: "bexp ⇒ ifexp"
fun if2bexp :: "ifexp ⇒ bexp"

and prove their correctness:

lemma "bval (if2bexp exp) s = ifval exp s"
lemma "ifval (b2ifexp exp) s = bval exp s"

Exercise 3.9. We define a new type of purely boolean expressions without any arithmetic

datatype pbexp =
  VAR vname | NOT pbexp | AND pbexp pbexp | OR pbexp pbexp

where variables range over values of type bool, as can be seen from the evaluation function:

fun pbval :: "pbexp ⇒ (vname ⇒ bool) ⇒ bool" where
"pbval (VAR x) s = s x" |
"pbval (NOT b) s = (¬ pbval b s)" |
"pbval (AND b1 b2) s = (pbval b1 s ∧ pbval b2 s)" |
"pbval (OR b1 b2) s = (pbval b1 s ∨ pbval b2 s)"

Define a function that checks whether a boolean expression is in NNF (negation normal form), i.e., if NOT is only applied directly to VARs:

fun is_nnf :: "pbexp ⇒ bool"

Now define a function that converts a bexp into NNF by pushing NOT inwards as much as possible:

fun nnf :: "pbexp ⇒ pbexp"

Prove that nnf does what it is supposed to do:

lemma pbval_nnf: "pbval (nnf b) s = pbval b s"
lemma is_nnf_nnf: "is_nnf (nnf b)"

An expression is in DNF (disjunctive normal form) if it is in NNF and if no OR occurs below an AND. Define a corresponding test:

fun is_dnf :: "pbexp ⇒ bool"

An NNF can be converted into a DNF in a bottom-up manner. The critical case is the conversion of AND b1 b2. Having converted b1 and b2, apply distributivity of AND over OR. If we write OR as a multi-argument function, we can express the distributivity step as follows: dist_AND (OR a1 ... an) (OR b1 ... bm) = OR (AND a1 b1) (AND a1 b2) ... (AND an bm). Define
fun dist_AND :: "pbexp ⇒ pbexp ⇒ pbexp"

and prove that it behaves as follows:

lemma pbval_dist: "pbval (dist_AND b1 b2) s = pbval (AND b1 b2) s"
lemma is_dnf_dist: "is_dnf b1 ⇒ is_dnf b2 ⇒ is_dnf (dist_AND b1 b2)"

Use dist_AND to write a function that converts an NNF to a DNF in the above bottom-up manner.

fun dnf_of_nnf :: "pbexp ⇒ pbexp"

Prove the correctness of your function:

lemma "pbval (dnf_of_nnf b) s = pbval b s"
lemma "is_nnf b ⇒ is_dnf (dnf_of_nnf b)"

Exercise 3.10. A stack underflow occurs when executing an ADD instruction on a stack of size less than two. In our semantics stack underflow leads to a term involving hd [], which is not an error or exception — HOL does not have those concepts — but some unspecified value. Modify theory ASM such that stack underflow is modelled by None and normal execution by Some, i.e., the execution functions have return type stack option. Modify all theorems and proofs accordingly. Hint: you may find split: option.split useful in your proofs.

Exercise 3.11. This exercise is about a register machine and compiler for aexp. The machine instructions are

datatype instr = LDI val reg | LD vname reg | ADD reg reg

where type reg is a synonym for nat. Instruction LDI i r loads i into register r, LD x r loads the value of x into register r, and ADD r1 r2 adds register r2 to register r1.

Define the execution of an instruction given a state and a register state; the result is the new register state:

type_synonym rstate = "reg ⇒ val"

fun exec1 :: "instr ⇒ state ⇒ rstate ⇒ rstate"

Define the execution exec of a list of instructions as for the stack machine.

The compiler takes an arithmetic expression a and a register r and produces a list of instructions whose execution places the value of a into r. The registers > r should be used in a stack-like fashion for intermediate results, the ones < r should be left alone. Define the compiler and prove it correct:

theorem "exec (comp a r) s rs r = aval a s"

Exercise 3.12. This exercise is a variation of the previous one with a different instruction set:

datatype instr0 = LDI0 val | LD0 vname | MV0 reg | ADD0 reg
All instructions refer implicitly to register 0 as a source or target: \textit{LDI}0 and \textit{LD}0 load a value into register 0, \textit{MV0} \textit{r} copies the value in register 0 into register \textit{r}, and \textit{ADD0} \textit{r} adds the value in register \textit{r} to the value in register 0; \textit{MV0} 0 and \textit{ADD0} 0 are legal. Define the execution functions

$$\text{fun exec01 :: "instr0 ⇒ state ⇒ rstate ⇒ rstate"}$$

and \text{exec0} for instruction lists.

The compiler takes an arithmetic expression \( a \) and a register \( r \) and produces a list of instructions whose execution places the value of \( a \) into register 0. The registers \( > r \) should be used in a stack-like fashion for intermediate results, the ones \( ≤ r \) should be left alone (with the exception of 0). Define the compiler and prove it correct:

$$\text{theorem "exec0 (comp0 a r) s rs 0 = aval a s"}$$
Chapter 4

Exercise 4.1. Start from the data type of binary trees defined earlier:

\[
\text{datatype } 'a \text{ tree} = \text{Tip} \mid \text{Node } '"a \text{ tree}' '"a \text{ tree}'
\]

An \textit{int tree} is ordered if for every \textit{Node l r} in the tree, \textit{l} and \textit{r} are ordered and all values in \textit{l} are \textit{< i} and all values in \textit{r} are \textit{> i}. Define a function that returns the elements in a tree and one the tests if a tree is ordered:

\[
\begin{align*}
\text{fun set :: } & "'a \text{ tree} \Rightarrow 'a \text{ set}"
\text{fun ord :: } & "\text{int tree} \Rightarrow \text{bool}"
\end{align*}
\]

Hint: use quantifiers.

Define a function \textit{ins} that inserts an element into an ordered \textit{int tree} while maintaining the order of the tree. If the element is already in the tree, the same tree should be returned.

\[
\begin{align*}
\text{fun ins :: } & "\text{int} \Rightarrow \text{int tree} \Rightarrow \text{int tree}"
\end{align*}
\]

Prove correctness of \textit{ins}:

\[
\begin{align*}
\text{lemma set_ins: } & \text{set (} \text{ins} \ x \ \text{t} \text{)} \text{ = } \{x\} \cup \text{set t}"
\text{theorem ord_ins: } & \text{ord t} \text{ = } \Rightarrow \text{ord (} \text{ins} \ x \ t \text{)"
\end{align*}
\]

Exercise 4.2. Formalize the following definition of palindromes

- The empty list and a singleton list are palindromes.
- If \textit{xs} is a palindrome, so is \textit{a # xs @ [a]}.

as an inductive predicate

\[
\begin{align*}
\text{inductive palindrome :: } & "'a \text{ list} \Rightarrow \text{bool}"
\end{align*}
\]

and prove

\[
\begin{align*}
\text{lemma } & \text{"palindrome xs } \Rightarrow \text{rev xs } = \text{xs"}
\end{align*}
\]

Exercise 4.3. We could also have defined \textit{star} as follows:

\[
\begin{align*}
\text{inductive star' :: } & "(\text{'}a \Rightarrow \text{'}a \Rightarrow \text{bool}) \Rightarrow \text{'}a \Rightarrow \text{'}a \Rightarrow \text{bool}" \text{ for } r \text{ where } \\
\text{refl': } & \text{"star' } r \ x \ x" \ \\
\text{step': } & \text{"star' } r \ x \ y \text{ } \Rightarrow \text{ } r \ y \ z \text{ } \Rightarrow \text{ } \text{star' } r \ x \ z"
\end{align*}
\]

The single \textit{r} step is performer after rather than before the \textit{star'} steps. Prove

\[
\begin{align*}
\text{lemma } & \text{"star' } r \ x \ y \text{ } \Rightarrow \text{ } \text{star } r \ x \ y" \ \\
\text{lemma } & \text{"star } r \ x \ y \text{ } \Rightarrow \text{ } \text{star' } r \ x \ y"
\end{align*}
\]

You may need lemmas. Note that rule induction fails if the assumption about the inductive predicate is not the first assumption.

Exercise 4.4. Analogous to \textit{star}, give an inductive definition of the \textit{n}-fold iteration of a relation \textit{r}: \textit{iter r n x y} should hold if there are \textit{x_0}, \ldots, \textit{x_n} such that \textit{x = x_0}, \textit{x_n = y} and \textit{r x_i x_{i+1}} for all \textit{i < n}:
Correct and prove the following claim:

**Lemma** "star r x y => iter r n x y"

**Exercise 4.5.** A context-free grammar can be seen as an inductive definition where each nonterminal A is an inductively defined predicate on lists of terminal symbols: A(w) means that w is in the language generated by A. For example, the production S -> aSb can be viewed as the implication S w => S (a # w @ [b]) where a and b are terminal symbols, i.e., elements of some alphabet. The alphabet can be defined as a datatype:

```plaintext
datatype alpha = asym | bsym
```

If you think of alpha.asym and bsym as "(" and ")", the following two grammars both generate strings of balanced parentheses (where ε is the empty word):

```plaintext
S  -> ε   | aSb   | SS
T  -> ε   | TaTb
```

Define them as inductive predicates and prove their equivalence:

**Inductive** S :: "alpha list => bool"

**Inductive** T :: "alpha list => bool"

**Lemma** TS: "T w => S w"

**Lemma** ST: "S w => T w"

**Corollary** SeqT: "S w <-> T w"

**Exercise 4.6.** In Chapter 3 we defined a recursive evaluation function `aval : aexp => state => val`. Define an inductive evaluation predicate and prove that it agrees with the recursive function:

**Inductive** `aval_rel : aexp => state => val => bool`

**Lemma** `aval_rel_aval`: "aval_rel a s v => aval a s = v"

**Lemma** `aval_aval_rel`: "aval a s = v => aval_rel a s v"

**Corollary** "aval_rel a s v <-> aval a s = v"

**Exercise 4.7.** Consider the stack machine from Chapter 3 and recall the concept of stack underflow from Exercise 3.10. Define an inductive predicate such that `ok n is n'` means that with any initial stack of length n the instructions can be executed without stack underflow and that the final stack has length n'.

Using the introduction rules for `ok`, prove the following special cases:

**Lemma** "ok 0 [LOAD x] (Suc 0)"

**Lemma** "ok 0 [LOAD x, LOADI v, ADD] (Suc 0)"

**Lemma** "ok (Suc (Suc 0)) [LOAD x, ADD, ADD, LOAD y] (Suc (Suc 0))"
Prove that $ok$ correctly computes the final stack size:

**lemma** "[$ok \ n \ is \ n'; \ length \ stk = n] \implies length (exec \ is \ s \ stk) = n'$

Prove that instruction sequences generated by $comp$ cannot cause stack underflow: $ok \ n \ (comp \ a) \ ?$ for some suitable value of $?$. 
Chapter 5

Exercise 5.1. Give a readable, structured proof of the following lemma:

**lemma** assumes $T$: "∀ $x$ $y$. $T$ $x$ $y$ ∨ $T$ $y$ $x$"

and $A$: "∀ $x$ $y$. $A$ $x$ $y$ ∧ $A$ $y$ $x$ → $x$ = $y$"

and $TA$: "∀ $x$ $y$. $T$ $x$ $y$ → $A$ $x$ $y$" and "$A$ $x$ $y$"

shows "$T$ $x$ $y$"

Each step should use at most one of the assumptions $T$, $A$ or $TA$.

Exercise 5.2. Give a readable, structured proof of the following lemma:

**lemma** "(∃ $ys$ $zs$. $xs$ = $ys$ @ $zs$ ∧ length $ys$ = length $zs$) ∨ (∃ $ys$ $zs$. $xs$ = $ys$ @ $zs$ ∧ length $ys$ = length $zs$ + 1)"

Hint: There are predefined functions take and const drop of type nat ⇒ 'a list ⇒ 'a list such that take $k$ [$x_1$,...] = [$x_1$,...,$x_k$] and drop $k$ [$x_1$,...] = [$x_{k+1}$,...]. Let sledgehammer find and apply the relevant take and drop lemmas for you.

Exercise 5.3. Give a structured proof by rule inversion:

**lemma** assumes $a$: "ev (Suc (Suc $n$))" shows "ev $n$"

Exercise 5.4. Give a structured proof by rule inversions:

**lemma** "¬ ev (Suc (Suc (Suc $0$)))"

If there are no cases to be proved you can close a proof immediately with qed.

Exercise 5.5. Recall predicate star from Section 4.5 and iter from Exercise 4.4.

**lemma** "iter $r$ $n$ $x$ $y$ = ⇒ star $r$ $x$ $y"

Prove this lemma in a structured style, do not just sledgehammer each case of the required induction.

Exercise 5.6. Define a recursive function

**fun** elements :: "'a list ⇒ 'a set"

that collects all elements of a list into a set. Prove

**lemma** "$x$ ∈ elements $xs$ =⇒ ∃ $ys$ $zs$. $xs$ = $ys$ @ $x$ # $zs$ ∧ $x$ ∉ elements $ys"
Chapter 7

Exercise 7.1. Define a function that computes the set of variables that are assigned to in a command:

\[
\text{fun assigned :: "com ⇒ vname set"}
\]

Prove that if some variable is not assigned to in a command, then that variable is never modified by the command:

\[
\text{lemma } [\langle c, s \rangle ⇒ t; x \notin \text{assigned } c ] \implies s x = t x
\]

Exercise 7.2. Define a recursive function that determines if a command behaves like SKIP and prove its correctness:

\[
\text{fun skip :: "com ⇒ bool"}
\]

\[
\text{lemma } \text{skip } c = \implies c \sim \text{SKIP}
\]

Exercise 7.3. Define a recursive function

\[
\text{fun deskip :: "com ⇒ com"}
\]

that eliminates as many SKIPs as possible from a command. For example:

\[
\text{deskip } (\text{SKIP};\;\text{WHILE } b \text{ DO } \langle x ::= a;\;\text{SKIP} \rangle) = \text{WHILE } b \text{ DO } x ::= a
\]

Prove its correctness by induction on \(c\):

\[
\text{lemma } \text{deskip } c \sim c
\]

Remember lemma \text{sim_while_cong} for the WHILE case.

Exercise 7.4. A small-step semantics for the evaluation of arithmetic expressions can be defined like this:

\[
\text{inductive astep :: "aexp × state ⇒ aexp ⇒ bool" [infix "⇝" 50] where}
\]

\[
\langle V x, s \rangle \rightsquigarrow N (s x) \mid
\]

\[
\langle \text{Plus } \langle N i \rangle \langle N j \rangle, s \rangle \rightsquigarrow N (i + j) \mid
\]

Complete the definition with two rules for Plus that model a left-to-right evaluation strategy: reduce the first argument with \(\rightsquigarrow\) if possible, reduce the second argument with \(\rightsquigarrow\) if the first argument is a number. Prove that each \(\rightsquigarrow\) step preserves the value of the expression:

\[
\text{lemma } \langle a, s \rangle \rightsquigarrow a' \implies \text{aval } a = \text{aval } a'
\]

\[
\text{proof } \text{(induction rule: astep.induct [split_format (complete)])}
\]

Do not use the case idiom but write down explicitly what you assume and show in each case: fix ... assume ... show ....

Exercise 7.5. Prove or disprove (by giving a counterexample):

\[
\text{lemma } \langle IF And b_1 b_2 \text{ THEN } c_1 \text{ ELSE } c_2 \sim \}
\]

\[
\text{IF } b_1 \text{ THEN IF } b_2 \text{ THEN } c_1 \text{ ELSE } c_2 \text{ ELSE } c_2,\}
\]

\[
\text{lemma } \langle WHILE And b_1 b_2 \text{ DO } c \sim \text{WHILE } b_1 \text{ DO WHILE } b_2 \text{ DO } c\rangle
\]
definition $\text{Or} :: \text{bexp} \Rightarrow \text{bexp} \Rightarrow \text{bexp}$ where
\[ \text{Or } b_1 \ b_2 = \text{Not} \ (\text{And} \ (\text{Not} \ b_1) \ (\text{Not} \ b_2)) \]
lemma "WHILE Or $b_1 \ b_2$ DO $c \sim$
\[ \text{WHILE Or } b_1 \ b_2 \text{ DO } c ;; \text{WHILE } b_1 \text{ DO } c \]

Exercise 7.6. Define a new loop construct $\text{DO } c \text{ WHILE } b$ (where $c$ is executed once before $b$ is tested) in terms of the existing constructs in $\text{com}$:

definition $\text{Do} :: \text{com} \Rightarrow \text{bexp} \Rightarrow \text{com}$
\[ \text{DO } _\text{WHILE } _\text{}_\text{0}, 61 \text{ ~}[61] \text{ ~61} \]

Define a translation on commands that replaces all $\text{WHILE } b \text{ DO } c$ by suitable commands that use $\text{ DO } c \text{ WHILE } b$ instead:

definition $\text{dewhile} :: \text{com} \Rightarrow \text{com}$

Prove that your translation preserves the semantics:

lemma "dewhile $c \sim c$"

Exercise 7.7. Let $C :: \text{nat} \Rightarrow \text{com}$ be an infinite sequence of commands and $S :: \text{nat} \Rightarrow \text{com}$ an infinite sequence of states such that $C 0 = c ;; d$ and \[ \forall n. \ (C n, S n) \rightarrow (C (\text{Suc } n), S (\text{Suc } n)) \]. Then either all $C n$ are of the form $c_n ;; d$ and it is always $c_n$ that is reduced or $c_n$ eventually becomes $\text{SKIP}$.

Prove

lemma assumes "$C 0 = c ;; d$" and "\[ \forall n. \ (C n, S n) \rightarrow (C(\text{Suc } n), S(\text{Suc } n)) \]"

shows "$\exists c_1 \ c_2. \ C n = c_1 ;; d \land C(\text{Suc } n) = c_2 ;; d \land (c_1, S n) \rightarrow (c_2, S(\text{Suc } n)))$
\[ \lor (\exists k. \ C k = \text{SKIP} ;; d) "\]

For the following exercises copy theories $\text{Com}$, $\text{Big}_\text{Step}$ and $\text{Small}_\text{Step}$ and modify them as required. Those parts of the theories that do not contribute to the results required in the exercise can be discarded. If there are multiple proofs of the same result, you may update any one of them.

Exercise 7.8. Extend IMP with a $\text{REPEAT } c \text{ UNTIL } b$ command by adding the constructor

$\text{Repeat com bexp } (\text{"(REPEAT } _\text{}/ \text{ UNTIL } _\text{)}} \text{ [0, 61] 61}$
to datatype $\text{com}$. Adjust the definitions of big-step and small-step semantics, the proof that the big-step semantics is deterministic and the equivalence proof between the two semantics.

Exercise 7.9. Extend IMP with a new command $c_1 \text{ OR } c_2$ that is a non-deterministic choice: it may execute either $c_1$ or $c_2$. Add the constructor

$\text{Or com com } (\text{"_ OR/ _" [60, 61] 60})$
to datatype $\text{com}$. Adjust the definitions of big-step and small-step semantics, prove $(c_1 \text{ OR } c_2) \sim (c_2 \text{ OR } c_1)$ and update the equivalence proof between the two semantics.
Exercise 7.10. Extend IMP with exceptions. Add two constructors \textit{THROW} and \textit{TRY} \(c_1\) \textit{CATCH} \(c_2\) to datatype \(com:\)

\[
\begin{align*}
\text{THROW} & \mid \text{Try com com } ((\text{TRY } \_/ \text{CATCH } \_)) & \text{[0, 61] 61}
\end{align*}
\]

Command \textit{THROW} throws an exception. The only command that can catch an exception is \textit{TRY} \(c_1\) \textit{CATCH} \(c_2\): if an exception is thrown by \(c_1\), execution continues with \(c_2\), otherwise \(c_2\) is ignored. Adjust the definitions of big-step and small-step semantics as follows.

The big-step semantics is now of type \(com \times state \Rightarrow com \times state\). In a big step \((c,s) \Rightarrow (x,t), x\) can only be \textit{SKIP} (signalling normal termination) or \textit{THROW} (signalling that an exception was thrown but not caught).

The small-step semantics is of the same type as before. There are two final configurations now, \((\text{SKIP}, t)\) and \((\text{THROW}, t)\). Exceptions propagate upwards until an enclosing handler is found. That is, until a configuration \((\text{TRY THROW CATCH c, s})\) is reached and \textit{THROW} can be caught.

Adjust the equivalence proof between the two semantics such that you obtain \(cs \Rightarrow (\text{SKIP}, t) \iff cs \rightarrow^* (\text{SKIP}, t)\) and \(cs \Rightarrow (\text{THROW}, t) \iff cs \rightarrow^* (\text{THROW}, t)\). Also revise the proof of \((\exists cs'. cs \Rightarrow cs') \iff (\exists cs'. cs \rightarrow^* cs' \land \text{final cs}')\).
Chapter 8

For the following exercises copy and adjust theory Compiler. Intrepid readers only should attempt to adjust theory Compiler2 too.

Exercise 8.1. A common programming idiom is IF b THEN c, i.e., the ELSE-branch is a SKIP command. Look at how, for example, the command IF Less (V ‘z’) (N 5) THEN ‘y’ := N 3 ELSE SKIP is compiled by ccomp and identify a possible compiler optimization. Modify the definition of ccomp such that it generates fewer instructions for commands of the form IF b THEN c ELSE SKIP. Ideally the proof of theorem ccomp_bigstep should still work; otherwise adapt it.

Exercise 8.2. Building on Exercise 7.8, extend the compiler ccomp and its correctness theorem ccomp_bigstep to REPEAT loops. Hint: the recursion pattern of the big-step semantics and the compiler for REPEAT should match.

Exercise 8.3. Modify the machine language such that instead of variable names to values, the machine state maps addresses (integers) to values. Adjust the compiler and its proof accordingly.

In the simple version of this exercise, assume the existence of a globally bijective function addr_of with bij addr_of to adjust the compiler. Use the find_theorems search to find applicable theorems for bijective functions.

For the more advanced version and a slightly larger project, only assume that the function works on a finite set of variables: those that occur in the program. For the other, unused variables, it should return a suitable default address. In this version, you may want to split the work into two parts: first, update the compiler and machine language, assuming the existence of such a function and the (partial) inverse it provides. Second, separately construct this function from the input program, having extracted the properties needed for it in the first part. In the end, rearrange your theory file to combine both into a final theorem.

Exercise 8.4. This is a slightly more challenging project. Based on Exercise 8.3, and similarly to Exercise 3.11 and Exercise 3.12, define a second machine language that does not possess a built-in stack, but instead, in addition to the program counter, a stack pointer register. Operations that previously worked on the stack now work on memory, accessing locations based on the stack pointer.

For instance, let (pc, s, sp) be a configuration of this new machine consisting of program counter, store, and stack pointer. Then the configuration after an ADD instruction is (pc + 1, s(sp + 1 := s(sp + 1) + s sp), sp + 1), that is, ADD dereferences the memory at sp + 1 and sp, adds these two values and stores them at sp + 1, updating the values on the stack. It also increases the stack pointer by one to pop one value from the stack and leave the result at the top of the stack. This means the stack grows downwards.

Modify the compiler from Exercise 8.3 to work on this new machine language. Reformulate and reprove the easy direction of compiler correctness.
**Hint:** Let the stack start below 0, growing downwards, and use type \textit{nat} for addressing variable in \textit{LOAD} and \textit{STORE} instructions, so that it is clear by type that these instructions do not interfere with the stack.

**Hint:** When the new machine pops a value from the stack, this now unused value is left behind in the store. This means, even after executing a purely arithmetic expression, the values in initial and final stores are not all equal. But: they are equal above a given address. Define an abbreviation for this concept and use it to express the intermediate correctness statements.
Chapter 9

Exercise 9.1. Reformulate the inductive predicates \( \Gamma \vdash a : \tau \), \( \Gamma \vdash b \) and \( \Gamma \vdash c \) as three recursive functions

fun atype :: "tyenv ⇒ aexp ⇒ ty option"
fun bok :: "tyenv ⇒ bexp ⇒ bool"
fun cok :: "tyenv ⇒ com ⇒ bool"

and prove

lemma atyping_atype: "(\( \Gamma \vdash a : \tau \)) = (atype \( \Gamma \) a = Some \( \tau \))"
lemma btyping_bok: "(\( \Gamma \vdash b \)) = bok \( \Gamma \) b"
lemma ctyping_cok: "(\( \Gamma \vdash c \)) = cok \( \Gamma \) c"

Exercise 9.2. Modify the evaluation and typing of \( aexp \) by allowing \( \text{ints} \) to be coerced to \( \text{reals} \) with the predefined coercion function \( \text{real} :: \text{int} ⇒ \text{real} \) where necessary. Now every \( aexp \) has a value. Define an evaluation function:

fun aval :: "aexp ⇒ state ⇒ val"

Similarly, every \( aexp \) has a type. Define a function that computes the type of an \( aexp \)

fun atyp :: "tyenv ⇒ aexp ⇒ ty"

and prove that it computes the correct type:

lemma "\( \Gamma \vdash s ⇒ atyp \Gamma a = type \ (aval a s) \)"

Note that Isabelle inserts the coercion \( \text{real} \) automatically. For example, if you write \( Rv \ (i + r) \) where \( i :: \text{int} \) and \( r :: \text{real} \) then it becomes \( Rv \ (\text{real} \ i + x) \).

Exercise 9.3. Add a \( REPEAT \) loop (see Exercise 7.8) to the typed version of IMP and update the type soundness proof.

Exercise 9.4. Modify the typed version of IMP as follows. Values are now either integers or booleans. Thus variables can have boolean values too. Merge the two expressions types \( aexp \) and \( bexp \) into one new type \( \text{exp} \) of expressions that has the constructors of both types (of course without real constants). Combine \( \text{taval} \) and \( \text{tbval} \) into one evaluation predicate \( \text{eval} :: \exp ⇒ \text{state} ⇒ \text{val} ⇒ \text{bool} \). Similarly combine the two typing predicates into one: \( \Gamma \vdash e : \tau \) where \( e :: \exp \) and the IMP-type \( \tau \) can be one of \( \text{Ity} \) or \( \text{Bty} \). Adjust the small-step semantics and the type soundness proof.

Exercise 9.5. Reformulate the inductive predicate \( \text{sec_type} \) as a recursive function and prove the equivalence of the two formulations:

fun ok :: "level ⇒ com ⇒ bool"

theorem "(\( l \vdash c \)) = ok \( l \) \( c \)"

Try to reformulate the bottom-up system \( \vdash c : l \) as a function that computes \( l \) from \( c \). What difficulty do you face?
Exercise 9.6. Define a bottom-up termination insensitive security type system 
\[ \Gamma \vdash c : l \] with subsumption rule:

\[ \text{inductive } \text{sec\_type}^2 : \text{"com} \Rightarrow \text{level} \Rightarrow \text{bool} \text{" } (\"(\Gamma \vdash c : l)\" 0,0 50) \]

Prove equivalence with the bottom-up system \( \Gamma \vdash c : l \) without subsumption rule:

\[ \text{lemma } \"\Gamma \vdash c : l \Rightarrow \Gamma \vdash' c : l\" \]

\[ \text{lemma } \"\Gamma \vdash c : l \Rightarrow \exists l' \geq l. \Gamma \vdash c : l'\" \]

Exercise 9.7. Define a function that erases those parts of a command that
contain variables above some security level:

\[ \text{fun } \text{erase} : \text{"level} \Rightarrow \text{com} \Rightarrow \text{com} \]

Function \text{erase} \( l \) should replace all assignments to variables with security level
\( \geq l \) by \text{SKIP}. It should also erase certain \text{IF}s and \text{WHILE}s, depending on the
security level of the boolean condition. Now show that \( c \) and \text{erase} \( l \) \( c \) behave
the same on the variables up to level \( l \):

\[ \text{theorem } \"[ [ (c,s) \Rightarrow s'; (\text{erase} \ l \ c, t) \Rightarrow t'; 0 \vdash c; s = t (\ < \ l) ] \]
\[ \Rightarrow s' = t' (\ < \ l)\" \]

This theorem looks remarkably like the noninterference lemma from theory
\text{Sec\_Typing} (although \( \leq \) has been replaced by \( < \)). You may want to start with
that proof and modify it. The structure should remain the same. You may also
need one or two simple additional lemmas.

In the theorem above we assume that both \( (c, s) \) and \( (\text{erase} \ l \ c, t) \) terminate. How about the following two properties:

\[ \text{lemma } \"[ [ (c,s) \Rightarrow s'; 0 \vdash c; s = t (\ < \ l) ]
\[ \Rightarrow \exists t'. (\text{erase} \ l \ c, t) \Rightarrow t' \land s' = t' (\ < \ l)\" \]

\[ \text{lemma } \"[ [ (\text{erase} \ l \ c, s) \Rightarrow s'; 0 \vdash c; s = t (\ < \ l) ]
\[ \Rightarrow \exists t'. (c,t) \Rightarrow t' \land s' = t' (\ < \ l)\" \]

Give proofs or counterexamples.
Chapter 10

Exercise 10.1. Define the definite initialisation analysis as two recursive functions

\[
\text{fun ivars :: "com ⇒ vname set"}
\]

\[
\text{fun ok :: "vname set ⇒ com ⇒ bool"}
\]

such that \( \text{ivars} \) computes the set of definitely initialised variables and \( \text{ok} \) checks that only initialised variable are accessed. Prove

\[
\text{lemma } "D A c A' \Rightarrow A' = A \cup \text{ivars} c \wedge \text{ok} A c"
\]

\[
\text{lemma } "\text{ok} A c \Rightarrow D A c (A \cup \text{ivars} c)"
\]

Exercise 10.2. Extend \( \text{afold} \) with simplifying addition of 0. That is, for any expression \( c, e+0 \) and \( 0+e \) should be simplified to just \( c \), including the case where the 0 is produced by knowledge of the content of variables.

\[
\text{fun afold :: "aexp ⇒ tab ⇒ aexp"}
\]

Re-prove the results in this section with the extended version by copying and adjusting the contents of theory \( \text{Fold} \).

\[
\text{theorem } "\text{fold} c \text{ empty} \sim c"
\]

Exercise 10.3. Strengthen and re-prove the congruence rules for conditional semantic equivalence to take the value of boolean expressions into account in the IF and WHILE cases. As a reminder, the weaker rules are:

\[
[P \models b \leftrightarrow b'; P \models c \sim c'; P \models d \sim d']
\implies P \models \text{IF b THEN c ELSE d} \sim \text{IF b' THEN c' ELSE d'}
\]

\[
[P \models b \leftrightarrow b'; P \models c \sim c'; P \models d \sim d' \wedge s \sim s'. ] [(c, s) ⇒ s'; P s; \text{bval b s}] \implies P s'
\implies P \models \text{WHILE b DO c} \sim \text{WHILE b' DO c'}
\]

Find a formulation that takes \( b \) into account for equivalences about \( c \) or \( d \).

Exercise 10.4. Extend constant folding with analysing boolean expressions and eliminate dead IF branches as well as loops whose body is never executed. Use the contents of theory \( \text{Fold} \) as a blueprint.

\[
\text{fun bfold :: "bexp ⇒ tab ⇒ bexp"}
\]

\[
\text{primrec bdefs :: "com ⇒ tab ⇒ tab"}
\]

\[
\text{primrec fold' :: "com ⇒ tab ⇒ com"}
\]

Hint: you will need to make use of stronger congruence rules for conditional semantic equivalence.

\[
\text{lemma fold'_equiv: "approx t \models c \sim fold' c t"}
\]

\[
\text{theorem constant_folding_equiv': "fold' c empty \sim c"}
\]
Exercise 10.5. This exercise builds infrastructure for Exercise 10.6, where we will have to manipulate partial maps from variable names to variable names.

**type_synonym** tab = "vname ⇒ vname option"

In addition to the function `merge` from theory `Fold`, implement two functions `remove` and `remove_all` that remove one variable name from the range of a map, and a set of variable names from the domain and range of a map.

**definition** remove :: "vname ⇒ tab ⇒ tab"

**definition** remove_all :: "vname set ⇒ tab ⇒ tab"

Prove the following lemmas.

**lemma** "ran (remove x t) = ran t − {x}"

**lemma** "ran (remove_all S t) ⊆ −S"

**lemma** "dom (remove_all S t) ⊆ −S"

**lemma** "remove_all {x} (t (x:= y)) = remove_all {x} t"

**lemma** "remove_all {x} (remove x t) = remove_all {x} t"

**lemma** "remove_all A (remove_all B t) = remove_all (A ∪ B) t"

**lemma** merge_remove_all:
  assumes "remove_all S t1 = remove_all S t"
  assumes "remove_all S t2 = remove_all S t"
  shows "remove_all S (merge t1 t2) = remove_all S t"

Exercise 10.6. This is a more challenging exercise. Define and prove correct copy propagation. Copy propagation is similar to constant folding, but propagates the right-hand side of assignments if these right-hand sides are just variables. For instance, the program `x := y; z := x + z` will be transformed into `x := y; z := y + z`. The assignment `x := y` can then be eliminated in a liveness analysis. Copy propagation is useful for cleaning up after other optimisation phases.

To do this, take the definitions for constant folding from theory `Fold` and adjust them to do copy propagation instead (without constant folding). Use the functions from Exercise 10.5 in your definition. The general proof idea and structure of constant folding remains applicable. Adjust it according to your new definitions.

**primrec** copy :: "com ⇒ tab ⇒ com"

**theorem** "copy c empty ~ c"

Exercise 10.7. Prove the following termination-insensitive version of the correctness of L:

**theorem** "[ (c,s) ⇒ s'; (c,t) ⇒ t'; s = t on L c X ] ⇒ s' = t' on X"

Do not derive it as a corollary of the original correctness theorem but prove it separately. Hint: modify the original proof.
Exercise 10.8. Find a command \( c \) such that \( \text{bury} (\text{bury} \ c \ []\ []) \neq \text{bury} \ c \ []. \) For an arbitrary command, can you put a limit on the amount of burying needed until everything that is dead is also buried?

Exercise 10.9. Let \( lvars \ c / rvars \ c \) be the set of variables that occur on the left-hand / right-hand side of an assignment in \( c \). Let \( rvars \ c \) additionally including those variables mentioned in the conditionals of \( \text{IF} \) and \( \text{WHILE} \). Both functions are predefined in theory \( \text{Vars} \). Show the following two properties of the small-step semantics. Variables that are not assigned to do not change their value:

lemma "\([[(c,s) \rightarrow (c',s')]; lvars \ c \cap X = []] \implies s = s' \ on X""

The reduction behaviour of a command is only influenced by the variables read by the command:

lemma "\([[(c,s) \rightarrow (c',s')]; s = t on X; rvars \ c \subseteq X] \implies \exists t'. (c,t) \rightarrow (c',t') \land s' = t' on X""

Hint: prove single step versions of the lemmas first.

Exercise 10.10. An available definitions analysis determines which previous assignments \( x := a \) are valid equalities \( x = a \) at later program points. For example, after \( x := y+1 \) the equality \( x = y+1 \) is available, but after \( x := y+1; y := 2 \) the equality \( x = y+1 \) is no longer available. The motivation for the analysis is that if \( x = a \) is available before \( v := a \) then \( v := x \) can be replaced by \( v := a \).

Define an available definitions analysis as a gen/kill analysis, for suitably defined \( \text{gen} \) and \( \text{kill} \) (which may need to be mutually recursive):

fun \( \text{gen} :: \ "\text{com} \Rightarrow (vname \ast aexp) \ set" \) and \( \ "\text{kill} :: \ "\text{com} \Rightarrow (vname \ast aexp) \ set" \ where \ "AD AD A c = \text{gen} c \cup (A \setminus \text{kill} c)" \)

The defining equations for both \( \text{gen} \) and \( \text{kill} \) follow the where and are separated by "|" as usual.

A call \( \text{AD} \ A \ c \) should compute the available definitions after the execution of \( c \) assuming that the definitions in \( A \) are available before the execution of \( c \).

Prove correctness of the analysis:

theorem "\([(c,s) \Rightarrow s'; \forall (x,a) \in A. \ s \ x = \text{aval} a \ s] \implies \forall (x,a) \in AD A c. \ s' \ x = \text{aval} a \ s'""

Exercise 10.13. In the context of ordinary live variable analysis, elimination of dead variables (bury) is not idempotent (Exercise 10.8). Now define the textually identical function \( \text{bury} \) in the context of true liveness analysis (theory \( \text{Live...True} \)) and prove that it is idempotent.

fun \( \text{bury} :: \ "\text{com} \Rightarrow vname \ set \Rightarrow \text{com}" \)
The following two tweaks improve proof automation:

```latex
declare L.simps(5)[simp]
lemmas L_mono2 = L_mono[unfolded mono_def]
```

To show that `bury` is idempotent we need a key lemma:

```latex
lemma L_bury: "X ⊆ Y \implies L (bury c Y) X = L c X"
```

The proof is straightforward except for the case `While b c` where reasoning about `lfp` is required. Sledgehammer should help with the details.

Now we can prove idempotence of `bury`, again by induction on `c`:

```latex
theorem bury_idemp: "bury (bury c X) X = bury c X"
```

Due to lemma `L_bury`, even the `While` case should be easy.
Chapter 11

Exercise 11.1. Building on Exercise 7.8, extend the denotational semantics and the equivalence proof with the big-step semantics with a REPEAT loop.

Exercise 11.2. Consider Example 11.14 and prove the following equation by induction on \( n \):

\[
\text{lemma } "((\ W (\lambda s. s 'x'' \neq 0) (([[s,t]. t = s('x'' := s 'x'' - 1)]]) \sim n) \{ = \{ (s,t). 0 \leq s 'x'' & s 'x'' < \text{int } n & t = s('x'' := 0)\}"
\]

Exercise 11.3. Consider Example 11.14 but with the loop condition \( b = \text{Less}(N \ 0) (V 'x'') \). Find a closed expression \( M \) (containing \( n \)) for \( f^n \{ \) and prove \( f^n \{ = M \).

Exercise 11.4. Define an operator \( B \) such that you can express the equation for \( D (\ IF b \ THEN c \ ELSE c_2) \) in a point free way.

\[
\text{definition } B :: "bexp \Rightarrow (state \times state) set"
\]

\[
\text{lemma } "D (IF b \ THEN c_1 \ ELSE c_2) = (B b O D c_1) \cup (B (Not b) O D c_2)"
\]

Similarly, find a point free equation for \( W (bval b) dc \) and use it to write down a point free version of \( D (\ WHILE b \ DO c) \) (still using \( lfp \)). Prove that your equations are equivalent to the old ones.

Exercise 11.5. Let the 'thin' part of a relation be its single-valued subset:

\[
\text{definition } thin :: "'a rel \Rightarrow 'a rel" \ where
\]

\[
"\text{thin } R = \{(a,b) . (a,b) \in R \land (\forall c. (a,c) \in R \Rightarrow c = b)\}"
\]

Prove

\[
\text{lemma } fixes f :: "'a rel \Rightarrow 'a rel" \ assumes " mono f" and thin_f: "\land R. f (\text{thin } R) \subseteq \text{thin } (f R)"
\]

\[
\text{shows " single_valued } (lfp f)"
\]

Exercise 11.6. Generalise our set-theoretic treatment of continuity and least fixpoints to chain-complete partial orders (cpos), i.e. partial orders \( \leq \) that have a least element \( \bot \) and where every chain \( c 0 \leq c 1 \leq \ldots \) has a least upper bound \( lub c \) where \( c :: nat \Rightarrow 'a \). This setting is described by the following type class \( \text{cpos} \) which is an extension of the type class \( \text{order} \) of partial orders. For details see the description of type classes in Chapter 13.

context \( \text{order} \)

begin

\[
\text{definition } chain :: "(nat \Rightarrow 'a) \Rightarrow bool" \ where
\]

"chain c = (\forall n. c n \leq c (Suc n))"

end
Give an example that the following stronger termination-sensitive property function and an assignment in "Dep and prove lub_ub and bot_least bot fixes class "Dep and lfp × where "Dep fun lub" (1 lub c) = lub (f o c). Prove that if f is monotone and continuous then lub (λn. (f^n) c) is the least (pre)fixpoint of f:

definition cont : "('a::cpo ⇒ 'b::cpo) ⇒ bool" where "cont f = (∀ c. chain c ⇒ f (lub c) = lub (f o c))"

abbreviation "fix f ≡ lub (λn. (f^n) bot)"

lemma fix_lfp: assumes "mono f" and "f p ≤ p" shows "fix f ≤ p"
theorem fix_fp: assumes "mono f" and "cont f" shows "f(fix f) = fix f"

Exercise 11.7. We define a dependency analysis Dep that maps commands to relations between variables such that (x, y) ∈ Dep c means that in the execution of c the initial value of x can influence the final value of y:

fun Dep :: "com ⇒ (vname + vname) set" where "Dep SKIP = Id" | "Dep (x::a) = {(u,v). if u = x then u ∈ vars a else u = v}" | "Dep (c1;;c2) = Dep c1 O Dep c2" | "Dep (IF b THEN c1 ELSE c2) = Dep c1 ∪ Dep c2 ∪ vars b × UNIV" | "Dep (WHILE b DO c) = lfp(λR. Id ∪ vars b × UNIV ∪ Dep c O R)"

where × is the cross product of two sets. Prove monotonicity of the function lfp is applied to.

For the correctness statement define

abbreviation Deps :: "com ⇒ vname set ⇒ vname set" where "Deps c X = (∪ x ∈ X. {y. (y,x) : Dep c})"

and prove

lemma "[(c,s) ⇒ s'; (c,t) ⇒ t'; s = t on Deps c X] ⇒ s' = t' on X"

Give an example that the following stronger termination-sensitive property

[(c, s) ⇒ s'; s = t on Deps c X] ⇒ ∃ t'. (c, t) ⇒ t' ∧ s' = t' on X

does not hold. Hint: X = {c}

In the definition of Dep (IF b THEN c1 ELSE c2) the variables in b can influence all variables (UNIV). However, if a variable is not assigned to in c1 and c2 it is not influenced by b (ignoring termination). Theory Vars defines a function lvars such that lvars c is the set of variables on the left-hand side of an assignment in c. Modify the definition of Dep as follows: replace UNIV by
lvars c_1 \cup lvars c_2 \text{ (in the case } IF \ b \ THEN \ c_1 \ ELSE \ c_2) \text{ and by } lvars \ c \text{ (in the case } WHILE \ b \ DO \ c). \text{ Adjust the proof of the above correctness statement.}
Chapter 12

Exercise 12.2. Define \( \text{bsubst} \) and prove the Substitution Lemma:

\[
\text{fun \ bsubst :: } "\text{bexp} \Rightarrow \text{aexp} \Rightarrow \text{vname} \Rightarrow \text{bexp}"
\]

\[
\text{lemma \ bsubstitution: } "\text{bval} (\text{bsubst} b a x) s = \text{bval} b (s[a/x])"
\]

This may require a similar definition and proof for \( \text{aexp} \).

Exercise 12.3. Define a command \( \text{cmax} \) that stores the maximum of the values of the IMP variables \( x \) and \( y \) in the IMP variable \( z \) and prove that \( \text{cmax} \) satisfies its specification:

\[
\text{abbreviation \ cmax :: com}
\]

\[
\text{lemma } \vdash \{ \lambda s. \text{True} \} \text{cmax} \{ \lambda s. s''z'' = \text{max} (s''x'') (s''y'')\}
\]

Function \( \text{max} \) is the predefined maximum function. Proofs about \( \text{max} \) are often automatic when simplifying with \( \text{max_def} \).

Exercise 12.4. Define an equality operation for arithmetic expressions

\[
\text{definition \ Eq :: } "\text{aexp} \Rightarrow \text{aexp} \Rightarrow \text{bexp}"
\]

such that

\[
\text{lemma \ bval_Eq[simp]: } "\text{bval} (\text{Eq} a1 a2) s = (\text{aval} a1 s = \text{aval} a2 s)"
\]

Prove the following variant of the summation command correct:

\[
\text{lemma } \vdash \{ \lambda s. s''x'' = i \land 0 \leq i \} \text{cmax} \{ \lambda s. s''y'' = y \land 0 \leq y \} \{ \lambda t. t''y'' = \text{sum} i \}
\]

Exercise 12.5. Prove that the following command computes \( y - x \) if \( 0 \leq x \):

\[
\text{lemma } \vdash \{ \lambda s. s''x'' = x \land s''y'' = y \land 0 \leq x \} \text{cmax} \{ \lambda t. t''y'' = y - x \}
\]

Exercise 12.6. Define and verify a command \( \text{cmult} \) that stores the product of \( x \) and \( y \) in \( z \) assuming \( 0 \leq y \):

\[
\text{abbreviation \ cmult :: com}
\]

\[
\text{lemma } \vdash \{ \lambda s. s''x'' = x \land s''y'' = y \land 0 \leq y \} \text{cmult} \{ \lambda t. t''z'' = x*y \}
\]

You may have to simplify with \( \text{algebra_simps} \) to deal with \( \ast \).
Exercise 12.7. The following command computes an integer approximation $r$ of the square root of $i \geq 0$, i.e. $r^2 \leq i < (r+1)^2$. Prove

\[
\begin{align*}
\text{lemma } & \\
\forall i. & \{ \text{\lambda s. s' = i} \land 0 \leq i \} \\
\text{ 'r'' := N 0; 'r2'' := N 1; } & \text{ WHILE (Not (Less (V 'x') (V 'r2'))) } \text{ DO ('r'' := Plus (V 'r') (N 1)); } \text{ 'r2'' := Plus (V 'r2') (Plus (Plus (V 'r') (V 'r')) (N 1))} \\
\{ \text{\lambda s. (s 'r') ^2 \leq i } & \land i < (s 'r' + 1) ^2 \}
\end{align*}
\]

Figure out how $r^2$ is related to $r$ before formulating the invariant. The proof may require simplification with algebra_simps and power2_eq_square.

Exercise 12.8. Prove by induction:

\[
\begin{align*}
\text{lemma } & \\
\forall P. & \{ P \} c \{ \text{As. True} \}
\end{align*}
\]

Exercise 12.9. Design and prove correct a forward assignment rule of the form $\vdash \{ P \} x := a \{ ? \}$ where $?$ is some suitable postcondition that depends on $P$, $x$ and $a$. Hint: $?$ may need $\exists$.

\[
\begin{align*}
\text{lemma } & \\
\forall P. & \{ P \} x := a \{ \text{Questionmark} \}
\end{align*}
\]

(In case you wonder if your Questionmark is strong enough: see Exercise 12.15)

Exercise 12.10. Prove

\[
\begin{align*}
\forall P. & \{ P \} c \{ Q \} \iff (\forall s. \text{P} s \rightarrow \text{wp c Q s})
\end{align*}
\]

Exercise 12.11. Replace the assignment command with a new command $Do f$ where $f :: \text{state} \Rightarrow \text{state}$ can be an arbitrary state transformer. Update the big-step semantics, Hoare logic and the soundness and completeness proofs.

Exercise 12.12. Which of the following rules are correct? Proof or counterexample!

\[
\begin{align*}
\text{lemma } & \\
\forall P. & \{ P \} c \{ Q \}; \vdash \{ P' \} c \{ Q' \} \Rightarrow \\
\vdash \{ \text{As. P s V P' s} \} c \{ \text{As. Q s V Q' s} \}
\end{align*}
\]

\[
\begin{align*}
\text{lemma } & \\
\forall P. & \{ P \} c \{ Q \}; \vdash \{ P' \} c \{ Q' \} \Rightarrow \\
\vdash \{ \text{As. P s V P' s} \} c \{ \text{As. Q s V Q' s} \}
\end{align*}
\]

\[
\begin{align*}
\text{lemma } & \\
\forall P. & \{ P \} c \{ Q \}; \vdash \{ P' \} c \{ Q' \} \Rightarrow \\
\vdash \{ \text{As. P s --> P' s} \} c \{ \text{As. Q s --> Q' s} \}
\end{align*}
\]

Exercise 12.13. Based on Exercise 7.9, extend Hoare logic and the soundness and completeness proofs with nondeterministic choice.

Exercise 12.14. Based on Exercise 7.8, extend Hoare logic and the soundness and completeness proofs with a $\text{REPEAT}$ loop. Hint: think of $\text{REPEAT c UNTIL b}$ as equivalent to $c;; \text{WHILE Not b DO c}$. 

Exercise 12.15. The dual of the weakest precondition is the strongest postcondition $sp$. Define $sp$ in analogy with $wp$ via the big-step semantics:

**definition** $sp :: "com ⇒ assn ⇒ assn"$

Prove that $sp$ really is the strongest postcondition:

**lemma** "($|=\{P\} \ c \ \{Q\} |\ (\forall s. \ sp \ c \ P \ s → Q \ s)$"

In analogy with the derived equations for $wp$ given in the text, give and prove equations for “calculating” $sp$ for three constructs: $sp \ (x ::= a) \ P = Q_1$, $sp \ (c_1 ; c_2) \ P = Q_2$, and $sp \ (IF \ b \ THEN \ c_1 \ ELSE \ c_2) \ P = Q_3$. The $Q_i$ must not involve the semantics and may only call $sp$ recursively on the subcommands $c_i$. Hint: $Q_1$ requires an existential quantifier.

Exercise 12.16. Let $asum \ i$ be the annotated command $y := 0; W$ where $W$ is defined in Example 12.7. Prove

**lemma** "$\vdash \{\lambda s. \ s' 'x' = i\} \ strip(asum \ i) \ \{\lambda s. \ s' 'y' = sum \ i\}"

with the help of corollary $vc\_sound'$.

Exercise 12.17. Solve exercises 12.4 to 12.7 using the VCG: for every Hoare triple $\vdash \{P\} \ c \ \{Q\}$ from one of those exercises define an annotated version $C$ of $c$ and prove $\vdash \{P\} \ strip \ C \ \{Q\}$ with the help of corollary $vc\_sound'$.

Exercise 12.18. Having two separate functions $pre$ and $vc$ is inefficient. When computing $vc$ one often needs to compute $pre$ too, leading to multiple traversals of many subcommands. Define an optimised function

**fun** $prevc :: "acom ⇒ assn ⇒ assn × bool"

that traverses the command only once. Prove

**lemma** "$prevc \ C \ Q = (pre \ C \ Q, \ vc \ C \ Q)"

Exercise 12.19. Design a VCG that computes post rather than preconditions. Start by solving Exercise 12.9. Now modify theory $VCG$ as follows. Instead of $pre$ define a function

**fun** $post :: "acom ⇒ assn ⇒ assn"

such that (with the exception of loops) $post \ C \ P$ is the strongest postcondition of $C$ w.r.t. the precondition $P$ (see also Exercise 12.15). Now modify $vc$ such that it uses $post$ instead of $pre$ and prove its soundness and completeness.

**fun** $vc :: "acom ⇒ assn ⇒ boolean"

**lemma** $vc\_sound$: "$vc \ C \ P \implies \vdash \{P\} \ strip \ C \ \{post \ C \ P\}"

**lemma** $vc\_complete$: "$\vdash \{P\} \ c \ \{Q\}

\implies \exists C. \ strip \ C = c \land \ vc \ C \ P \land (\forall s. \ post \ C \ P \ s → Q \ s)"

Exercise 12.20. Prove total correctness of the commands in exercises 12.4 to 12.7.
Exercise 12.21. Modify the VCG to take termination into account. First modify type \textit{acom} by annotating \textit{WHILE} with a measure function in addition to an invariant:

\begin{verbatim}
Awhile assn "state ⇒ nat" bezp acom
"(( _, _) / \textit{WHILE} _ / DO _)" [0, 0, 61] 61
\end{verbatim}

Functions \textit{strip} and \textit{pre} remain almost unchanged. The only significant change is in the \textit{WHILE} case for \textit{vc}. Modify the old soundness proof to obtain

\textbf{lemma} \textit{vc\_sound}: "\textit{vc C Q} \implies \vdash t \{\textit{pre C Q}\} \textit{strip C \{Q\}}"

You may need the combined soundness and completeness of $\vdash t: (\vdash t \{P\} \ c \ \{Q\}) = (\vdash t \{P\} \ c \ \{Q\})$
Chapter 13

Exercise 13.11. Take the Isabelle theories that define commands, big-step semantics, annotated commands and the collecting semantics and extend them with a nondeterministic choice construct. Start with Exercise 7.9 and extend type \texttt{com}, then extend type \texttt{acom} with a corresponding construct:

\[
\text{Or} \; "\texttt{acom}" \; "\texttt{acom}" \; \texttt{a} \quad ("\_ OR// \_//\_" \; [60, 61, 0] \; 60)
\]

Finally extend function \texttt{Step}. Update proofs as well. Hint: think of \texttt{OR} as a nondeterministic conditional without a test.

Exercise 13.12. Prove the following lemmas in a detailed and readable style:

\[
\begin{align*}
\text{lemma} & \quad \text{fixes } x_0 :: "\texttt{a} :: order" \\
& \quad \text{assumes } \forall x, y. \; x \leq y \Rightarrow f \; x \leq f \; y \quad \text{and} \quad f \; q \leq q \\
& \quad \text{shows } \forall i. \; (f \; ^{\downarrow} \; i) \; x_0 \leq q
\end{align*}
\]

\[
\begin{align*}
\text{lemma} & \quad \text{fixes } x_0 :: "\texttt{a} :: order" \\
& \quad \text{assumes } \forall x, y. \; x \leq y \Rightarrow f \; x \leq f \; y \\
& \quad \text{shows } \forall i. \; (f \; ^{\downarrow} \; i) \; x_0 \leq (f \; ^{\downarrow} \; (i+1)) \; x_0
\end{align*}
\]

Exercise 13.13. Let \texttt{a} be a complete lattice and let \texttt{f :: a \Rightarrow a} be a monotone function. Give a readable proof that if \texttt{P} is a set of pre-fixpoints of \texttt{f} then \texttt{\bigcap P} is also a pre-fixpoint of \texttt{f}:

\[
\begin{align*}
\text{lemma} & \quad \text{fixes } P :: "\texttt{a}::complete\_lattice\; set" \\
& \quad \text{assumes } \forall p \in P. \; f \; p \leq p \\
& \quad \text{shows } f \; (\bigcap P) \leq \bigcap P
\end{align*}
\]

Sledgehammer should give you a proof you can start from.

Exercise 13.16. Give a readable proof that if \texttt{\gamma :: a::lattice} \Rightarrow \texttt{b::lattice} is a monotone function, then \texttt{\gamma (a_1 \cap a_2) \leq \gamma a_1 \cap \gamma a_2}:

\[
\begin{align*}
\text{lemma} & \quad \text{fixes } \gamma :: "\texttt{a::lattice} \Rightarrow \texttt{b::lattice}" \\
& \quad \text{assumes } \texttt{mono} : \forall x, y. \; x \leq y \Rightarrow \gamma x \leq \gamma y \\
& \quad \text{shows } \forall a_1, a_2. \; \gamma (a_1 \cap a_2) \leq \gamma a_1 \cap \gamma a_2
\end{align*}
\]

Give an example of two lattices and a monotone \texttt{\gamma} where \texttt{\gamma a_1 \cap \gamma a_2 \leq \gamma (a_1 \cap a_2)} does not hold.

Exercise 13.17. Consider a simple sign analysis based on this abstract domain:

\[
\text{datatype } \texttt{sign} = \texttt{None} | \texttt{Neg} | \texttt{Pos0} | \texttt{Any}
\]

\[
\begin{align*}
\text{fun } \gamma :: "\texttt{sign} \Rightarrow \texttt{val\; set}" & \quad \text{where} \\
& \quad "\gamma \; \texttt{None} = \{\}" \mid \\
& \quad "\gamma \; \texttt{Neg} = \{i. \; i < 0\}" \mid \\
\end{align*}
\]
Define inverse analyses for "+" and "<" and prove the required correctness properties:

```plaintext
fun inv_plus' :: "sign ⇒ sign ⇒ sign ⇒ sign * sign"
lemma
"[[ inv_plus' a1 a2 = (a1',a2'); i1 ∈ γ a1; i2 ∈ γ a2; i1+i2 ∈ γ a ]
⇒ i1 ∈ γ a1' ∧ i2 ∈ γ a2'] "

fun inv_less' :: "bool ⇒ sign ⇒ sign ⇒ sign * sign"
lemma
"[[ inv_less' bv a1 a2 = (a1',a2'); i1 ∈ γ a1; i2 ∈ γ a2; (i1<i2) = bv ]
⇒ i1 ∈ γ a1' ∧ i2 ∈ γ a2'] "
```

For the ambitious: turn the above fragment into a full-blown abstract interpreter by replacing the interval analysis in theory Abs_Int2_iintl by a sign analysis.