Exercises for *Concrete Semantics*

Tobias Nipkow Gerwin Klein

February 29, 2016

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 "*sum n* = *n* ∗ (*n*+1) *div* 2"

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*"

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

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

fun *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

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

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

such that *intersperse a* $[x_1, ..., x_n] = [x_1, a, x_2, a, ..., a, x_n]$. 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*:

fun *itadd* :: "*nat* ⇒ *nat* ⇒ *nat*"

Tail-recursive means that in the recursive case, *itadd* needs to call itself directly: *itadd* (*Suc m*) *n* = *itadd* .... 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:

fun *nodes* :: "*tree0* ⇒ *nat*"

Consider the following recursive function:

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:

datatype *exp* = *Var* | *Const int* | *Add exp exp* | *Mult exp exp*

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

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:

fun *evalp* :: "*int list* ⇒ *int* ⇒ *int*"

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

fun *coeffs* :: "*exp* ⇒ *int list*"

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

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

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:

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):

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.

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.

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

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:

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:

lemma *subst_lemma*: "$aval\ (subst\ x\ a\ e)\ s = aval\ e\ (s(x := aval\ a\ s))$"

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

lemma "$aval\ a1\ s = aval\ a2\ s$
$\Longrightarrow aval\ (subst\ x\ a1\ e)\ s = aval\ (subst\ x\ a2\ e)\ s$"

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

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

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

Exercise 3.6. The following type adds a *LET* construct to arithmetic expressions:

datatype $lexp = Nl\ int \mid Vl\ vname \mid Plusl\ lexp\ lexp \mid LET\ vname\ lexp\ lexp$

The *LET* constructor introduces a local variable: the value of $LET\ x\ e_1\ e_2$ 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 $lval :: lexp \Rightarrow state \Rightarrow int$ that evaluates *lexp* expressions. Remember $s(x := i)$.

Define a conversion $inline :: lexp \Rightarrow aexp$. The expression $LET\ x\ e_1\ e_2$ 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 *inline* is correct w.r.t. evaluation.

Exercise 3.7. Show that equality and less-or-equal tests on *aexp* are definable

definition Le :: "$aexp \Rightarrow aexp \Rightarrow bexp$"
definition Eq :: "$aexp \Rightarrow aexp \Rightarrow bexp$"

and prove that they do what they are supposed to:

lemma *bval_Le*: "$bval\ (Le\ a1\ a2)\ s = (aval\ a1\ s \leqslant aval\ a2\ s)$"
lemma *bval_Eq*: "$bval\ (Eq\ a1\ a2)\ s = (aval\ a1\ s = aval\ a2\ s)$"

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 $ifval$ :: "$ifexp \Rightarrow state \Rightarrow bool$"

Then define two translation functions

fun $b2ifexp$ :: "$bexp \Rightarrow ifexp$"
fun $if2bexp$ :: "$ifexp \Rightarrow bexp$"

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 $pbval$ :: "$pbexp \Rightarrow (vname \Rightarrow bool) \Rightarrow bool$" **where**
"$pbval\ (VAR\ x)\ s = s\ x$" |
"$pbval\ (NOT\ b)\ s = (\neg\ pbval\ b\ s)$" |
"$pbval\ (AND\ b1\ b2)\ s = (pbval\ b1\ s \wedge pbval\ b2\ s)$" |
"$pbval\ (OR\ b1\ b2)\ s = (pbval\ b1\ s \vee pbval\ b2\ s)$"

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 :: "$pbexp \Rightarrow bool$"

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

fun nnf :: "$pbexp \Rightarrow pbexp$"

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 :: "$pbexp \Rightarrow bool$"

An NNF can be converted into a DNF in a bottom-up manner. The critical case is the conversion of $AND\ b_1\ b_2$. Having converted b_1 and b_2, apply distributivity of AND over OR. If we write OR as a multi-argument function, we can express the distributivity step as follows: $dist\_AND\ (OR\ a_1\ ...\ a_n)\ (OR\ b_1\ ...\ b_m) = OR\ (AND\ a_1\ b_1)\ (AND\ a_1\ b_2)\ ...\ (AND\ a_n\ b_m)$. Define

fun *dist_AND* :: "*pbexp* ⇒ *pbexp* ⇒ *pbexp*"

and prove that it behaves as follows:

lemma *pbval_dist*: "*pbval* (*dist_AND* *b*1 *b*2) *s* = *pbval* (*AND* *b*1 *b*2) *s*"
lemma *is_dnf_dist*: "*is_dnf* *b*1 ⟹ *is_dnf* *b*2 ⟹ *is_dnf* (*dist_AND* *b*1 *b*2)"

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* r_1 r_2 adds register r_2 to register r_1.

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 *exec*1 :: "*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 *instr*0 = *LDI*0 *val* | *LD*0 *vname* | *MV*0 *reg* | *ADD*0 *reg*

All instructions refer implicitly to register 0 as a source or target: $LDI0$ and $LD0$ load a value into register 0, $MV0\ r$ copies the value in register 0 into register r, and $ADD0\ r$ adds the value in register r to the value in register 0; $MV0\ 0$ and $ADD0\ 0$ are legal. Define the execution functions

fun *exec*01 :: "*instr*0 ⇒ *state* ⇒ *rstate* ⇒ *rstate*"

and *exec*0 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 $\leqslant r$ should be left alone (with the exception of 0). Define the compiler and prove it correct:

theorem "*exec*0 (*comp*0 *a r*) *s rs* 0 = *aval a s*"

Chapter 4

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

datatype *'a tree = Tip | Node "'a tree" 'a "'a tree"*

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

fun *set* :: *"'a tree ⇒ 'a set"*
fun *ord* :: *"int tree ⇒ bool"*

Hint: use quantifiers.

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

fun *ins* :: *"int ⇒ int tree ⇒ int tree"*

Prove correctness of *ins*:

lemma *set_ins*: *"set(ins x t) = {x} ∪ set t"*
theorem *ord_ins*: *"ord t ⟹ ord(ins i t)"*

Exercise 4.2. Formalize the following definition of palindromes

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

as an inductive predicate

inductive *palindrome* :: *"'a list ⇒ bool"*

and prove

lemma *"palindrome xs ⟹ rev xs = xs"*

Exercise 4.3. We could also have defined *star* as follows:

inductive *star'* :: *"('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool"* **for** *r* **where**
refl': *"star' r x x"* |
step': *"star' r x y ⟹ r y z ⟹ star' r x z"*

The single r step is performer after rather than before the *star'* steps. Prove

lemma *"star' r x y ⟹ star r x y"*
lemma *"star r x y ⟹ star' r x y"*

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 *star*, give an inductive definition of the n-fold iteration of a relation r: *iter r n x y* should hold if there are $x_0, \ldots, x_n$ such that $x = x_0$, $x_n = y$ and $r\ x_i\ x_{i+1}$ for all $i < n$:

inductive $iter$ :: "$('a \Rightarrow 'a \Rightarrow bool) \Rightarrow nat \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$" **for** r **where**

Correct and prove the following claim:

lemma "$star\ r\ x\ y \Longrightarrow iter\ r\ n\ x\ y$"

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: $A(w)$ mans that w is in the language generated by A. For example, the production $S \to aSb$ can be viewed as the implication $S\ w \Longrightarrow S\ (a\ \#\ w\ @\ [b])$ where a and b are terminal symbols, i.e., elements of some alphabet. The alphabet can be defined as a datatype:

datatype $alpha = a \mid b$

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

$$\begin{array}{lllllll} S & \to & \varepsilon & \mid & aSb & \mid & SS \\ T & \to & \varepsilon & \mid & TaTb & & \end{array}$$

Define them as inductive predicates and prove their equivalence:

inductive S :: "$alpha\ list \Rightarrow bool$"
inductive T :: "$alpha\ list \Rightarrow bool$"
lemma TS: "$T\ w \Longrightarrow S\ w$"
lemma ST: "$S\ w \Longrightarrow T\ w$"
corollary $SeqT$: "$S\ w \longleftrightarrow T\ w$"

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

inductive $aval\_rel$ :: "$aexp \Rightarrow state \Rightarrow val \Rightarrow bool$"
lemma $aval\_rel\_aval$: "$aval\_rel\ a\ s\ v \Longrightarrow aval\ a\ s = v$"
lemma $aval\_aval\_rel$: "$aval\ a\ s = v \Longrightarrow aval\_rel\ a\ s\ v$"
corollary "$aval\_rel\ a\ s\ v \longleftrightarrow aval\ a\ s = v$"

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

inductive ok :: "$nat \Rightarrow instr\ list \Rightarrow nat \Rightarrow bool$"

such that $ok\ n\ is\ n'$ means that with any initial stack of length n the instructions is 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 "$[\![ok\ n\ is\ n';\ length\ stk = n]\!] \Longrightarrow length\ (exec\ is\ s\ stk) = n'$"

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: "$\forall x\ y.\ T\ x\ y \lor T\ y\ x$"
and A: "$\forall x\ y.\ A\ x\ y \land A\ y\ x \longrightarrow x = y$"
and TA: "$\forall x\ y.\ T\ x\ y \longrightarrow A\ x\ y$" **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 "$(\exists\, ys\ zs.\ xs = ys\ @\ zs \land length\ ys = length\ zs)$
$\lor (\exists\, ys\ zs.\ xs = ys\ @\ zs \land length\ ys = length\ zs + 1)$"

Hint: There are predefined functions *take* and const drop of type $nat \Rightarrow\ 'a\ list \Rightarrow\ 'a\ list$ such that $take\ k\ [x_1,\ldots] = [x_1,\ldots,x_k]$ and $drop\ k\ [x_1,\ldots] = [x_{k+1},\ldots]$. 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 "$\neg\ ev(Suc(Suc(Suc\ 0)))$"

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

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

lemma "$iter\ r\ n\ x\ y \Longrightarrow star\ r\ x\ y$"

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 $elems$ :: "$'a\ list \Rightarrow\ 'a\ set$"

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

lemma "$x \in elems\ xs \Longrightarrow \exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs \land x \notin elems\ ys$"

Exercise 5.7. Extend Exercise 4.5 with a function that checks if some *alpha list* is a balanced string of parentheses. More precisely, define a recursive function

fun $balanced$ :: "$nat \Rightarrow alpha\ list \Rightarrow bool$"

such that $balanced\ n\ w$ is true iff (informally) $a^n\ @\ w \in S$. Formally, prove

corollary "$balanced\ n\ w \longleftrightarrow S\ (replicate\ n\ a\ @\ w)$"

where $replicate :: nat \Rightarrow\ 'a \Rightarrow\ 'a\ list$ is predefined and $replicate\ n\ x$ yields the list $[x,\ \ldots,\ x]$ of length n.

Chapter 7

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

fun *assigned* :: "$com \Rightarrow vname\ set$"

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

lemma "$[\![\ (c,\ s) \Rightarrow t;\ x \notin assigned\ c\ ]\!] \Longrightarrow s\ x = t\ x$"

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

fun *skip* :: "$com \Rightarrow bool$"
lemma "$skip\ c \Longrightarrow c \sim SKIP$"

Exercise 7.3. Define a recursive function

fun *deskip* :: "$com \Rightarrow com$"

that eliminates as many *SKIP*s as possible from a command. For example:

$deskip\ (SKIP;;\ WHILE\ b\ DO\ (x ::= a;;\ SKIP)) = WHILE\ b\ DO\ x ::= a$

Prove its correctness by induction on c:

lemma "$deskip\ c \sim c$"

Remember lemma *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:

inductive *astep* :: "$aexp \times state \Rightarrow aexp \Rightarrow bool$" (**infix** "$\rightsquigarrow$" 50) **where**
"$(V\ x,\ s) \rightsquigarrow N\ (s\ x)$" |
"$(Plus\ (N\ i)\ (N\ j),\ s) \rightsquigarrow N\ (i + j)$" |

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:

lemma "$(a,\ s) \rightsquigarrow a' \Longrightarrow aval\ a\ s = aval\ a'\ s$"
proof (*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):

lemma "$IF\ And\ b_1\ b_2\ THEN\ c_1\ ELSE\ c_2 \sim$
$IF\ b_1\ THEN\ IF\ b_2\ THEN\ c_1\ ELSE\ c_2\ ELSE\ c_2$"
lemma "$WHILE\ And\ b_1\ b_2\ DO\ c \sim WHILE\ b_1\ DO\ WHILE\ b_2\ DO\ c$"

definition Or :: "$bexp \Rightarrow bexp \Rightarrow bexp$" **where**
"$Or\ b_1\ b_2 = Not\ (And\ (Not\ b_1)\ (Not\ b_2))$"
lemma "$WHILE\ Or\ b_1\ b_2\ DO\ c \sim$
$WHILE\ Or\ b_1\ b_2\ DO\ c;;\ WHILE\ b_1\ DO\ c$"

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

definition Do :: "$com \Rightarrow bexp \Rightarrow com$" ("$DO$ _ $WHILE$ _" [0, 61] 61)

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

fun $dewhile$ :: "$com \Rightarrow com$"

Prove that your translation preserves the semantics:

lemma "$dewhile\ c \sim c$"

Exercise 7.7. Let $C :: nat \Rightarrow com$ be an infinite sequence of commands and $S :: nat \Rightarrow state$ an infinite sequence of states such that $C\ 0 = c;;\ d$ and $\forall n.\ (C\ n,\ S\ n) \rightarrow (C\ (Suc\ n),\ S\ (Suc\ n))$. 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 $SKIP$. Prove

lemma assumes "$C\ 0 = c;;d$" **and** "$\forall n.\ (C\ n,\ S\ n) \rightarrow (C(Suc\ n),\ S(Suc\ n))$"
shows "$(\forall n.\ \exists c_1\ c_2.\ C\ n = c_1;;d \wedge C(Suc\ n) = c_2;;d \wedge$
$(c_1,\ S\ n) \rightarrow (c_2,\ S(Suc\ n)))$
$\vee (\exists k.\ C\ k = SKIP;;d)$"

For the following exercises copy theories Com, Big_Step and $Small\_Step$ 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 $REPEAT\ c\ UNTIL\ b$ command by adding the constructor

```
Repeat com bexp   ("(REPEAT _/ UNTIL _)"  [0, 61] 61)
```

to datatype 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 $c_1\ OR\ c_2$ that is a nondeterministic choice: it may execute either c_1 or c_2. Add the constructor

```
Or com com   ("_ OR/ _" [60, 61] 60)
```

to datatype com. Adjust the definitions of big-step and small-step semantics, prove $(c_1\ OR\ c_2) \sim (c_2\ OR\ c_1)$ and update the equivalence proof between the two semantics.

Exercise 7.10. Extend IMP with exceptions. Add two constructors *THROW* and *TRY* c_1 *CATCH* c_2 to datatype *com*:

```
THROW  |  Try com com   ("(TRY _/ CATCH _)"  [0, 61] 61)
```

Command *THROW* throws an exception. The only command that can catch an execption is *TRY* c_1 *CATCH* c_2: if an execption 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 *SKIP* (signalling normal termination) or *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, $(SKIP,\ t)$ and $(THROW,\ t)$. Exceptions propagate upwards until an enclosing handler is found. That is, until a configuration $(TRY\ THROW\ CATCH\ c,\ s)$ is reached and *THROW* can be caught.

Adjust the equivalence proof between the two semantics such that you obtain $cs \Rightarrow (SKIP,t) \longleftrightarrow cs \rightarrow\!* (SKIP,t)$ and $cs \Rightarrow (THROW,t) \longleftrightarrow cs \rightarrow\!* (THROW,t)$. Also revise the proof of $(\exists\, cs'.\ cs \Rightarrow cs') \longleftrightarrow (\exists\, cs'.\ cs \rightarrow\!* cs' \wedge \mathit{final}\ cs')$.

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* $(V\ ''x'')\ (N\ 5)$ *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 bijectivte 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 you 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 *nat* for addressing variable in *LOAD* and *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 *int*s to be coerced to *real*s with the predefined coercion function *real_of_int* :: *int* ⇒ *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 "$\Gamma \vdash s \Longrightarrow atyp\ \Gamma\ a = type\ (aval\ a\ s)$"

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

For the following two exercises copy theory *Types* and modify it as required.

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 *exp* of expressions that has the constructors of both types (of course without real constants). Combine *taval* and *tbval* into one evaluation predicate *eval* :: *exp* ⇒ *state* ⇒ *val* ⇒ *bool*. Similarly combine the two typing predicates into one: $\Gamma \vdash e : \tau$ where e :: *exp* and the IMP-type τ can be one of *Ity* or *Bty*. Adjust the small-step semantics and the type soundness proof.

Exercise 9.5. Reformulate the inductive predicate *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 $\vdash' c : l$ with subsumption rule:

inductive *sec_type2'* :: "*com* ⇒ *level* ⇒ *bool*" ("($\vdash'$ _ : _)" [0,0] 50)

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

lemma "$\vdash c : l \Longrightarrow \vdash' c : l$"
lemma "$\vdash' c : l \Longrightarrow \exists l' \geqslant l.\ \vdash c : l'$"

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

fun *erase* :: "*level* ⇒ *com* ⇒ *com*"

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

theorem "$[\![\ (c,s) \Rightarrow s';\ \ (erase\ l\ c,t) \Rightarrow t';\ \ 0 \vdash c;\ \ s = t\ (< l)\ ]\!]$
$\Longrightarrow s' = t'\ (< l)$"

This theorem looks remarkably like the noninterference lemma from theory *Sec_Typing* (although $\leqslant$ 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 $(erase\ l\ c,\ t)$ terminate. How about the following two properties:

lemma "$[\![\ (c,s) \Rightarrow s';\ \ 0 \vdash c;\ \ s = t\ (< l)\ ]\!]$
$\Longrightarrow \exists t'.\ (erase\ l\ c,\ t) \Rightarrow t' \wedge s' = t'\ (< l)$"

lemma "$[\![\ (erase\ l\ c,s) \Rightarrow s';\ \ 0 \vdash c;\ \ s = t\ (< l)\ ]\!]$
$\Longrightarrow \exists t'.\ (c,t) \Rightarrow t' \wedge s' = t'\ (< l)$"

Give proofs or counterexamples.

Chapter 10

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

fun $ivars$ :: "$com \Rightarrow vname\ set$"
fun ok :: "$vname\ set \Rightarrow com \Rightarrow bool$"

such that $ivars$ computes the set of definitely initialised variables and ok checks that only initialised variable are accessed. Prove

lemma "$D\ A\ c\ A' \Longrightarrow A' = A \cup ivars\ c \wedge ok\ A\ c$"
lemma "$ok\ A\ c \Longrightarrow D\ A\ c\ (A \cup ivars\ c)$"

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

fun $afold$ :: "$aexp \Rightarrow tab \Rightarrow aexp$"

Re-prove the results in this section with the extended version by copying and adjusting the contents of theory *Fold*.

theorem "$fold\ c\ empty \sim c$"

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:

$$[\![P \models b <\sim> b';\ P \models c \sim c';\ P \models d \sim d']\!]$$
$$\Longrightarrow P \models IF\ b\ THEN\ c\ ELSE\ d \sim IF\ b'\ THEN\ c'\ ELSE\ d'$$

$$[\![P \models b <\sim> b';\ P \models c \sim c';\ \bigwedge s\ s'.\ [\![(c,\ s) \Rightarrow s';\ P\ s;\ bval\ b\ s]\!] \Longrightarrow P\ s']\!]$$
$$\Longrightarrow P \models WHILE\ b\ DO\ c \sim WHILE\ b'\ DO\ c'$$

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 *Fold* as a blueprint.

fun $bfold$ :: "$bexp \Rightarrow tab \Rightarrow bexp$"
primrec $bdefs$ :: "$com \Rightarrow tab \Rightarrow tab$"
primrec $fold'$ :: "$com \Rightarrow tab \Rightarrow com$"

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

lemma $fold'\_equiv$: "$approx\ t \models c \sim fold'\ c\ t$"
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 "$ran\ (remove\_all\ S\ t) \subseteq -S$"
lemma "$dom\ (remove\_all\ S\ t) \subseteq -S$"
lemma "$remove\_all\ \{x\}\ (t\ (x{:=}\ y)) = remove\_all\ \{x\}\ t$"
lemma "$remove\_all\ \{x\}\ (remove\ x\ t) = remove\_all\ \{x\}\ t$"
lemma "$remove\_all\ A\ (remove\_all\ B\ t) = remove\_all\ (A \cup B)\ t$"
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 "$copy\ c\ empty \sim c$"

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

theorem "$[\![\ (c,s) \Rightarrow s';\ \ (c,t) \Rightarrow t';\ \ s = t\ on\ L\ c\ X\ ]\!] \Longrightarrow s' = t'\ on\ X$"

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 $bury\ (bury\ c\ \{\})\ \{\} \neq bury\ c\ \{\}$. 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 IF and $WHILE$. Both functions are predefined in theory $Vars$. Show the following two properties of the small-step semantics. Variables that are not assigned to do not change their value:

lemma $"[\![(c,s) \rightarrow\!* (c',s');\ lvars\ c \cap X = \{\}]\!] \Longrightarrow s = s'\ on\ X"$

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

lemma $"[\![(c,s) \rightarrow\!* (c',s');\ s = t\ on\ X;\ rvars\ c \subseteq X]\!]$
$\Longrightarrow \exists t'.\ (c,t) \rightarrow\!* (c',t') \wedge s' = t'\ on\ X"$

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 := a` can be replaced by `v := x`.

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

fun gen :: $"com \Rightarrow (vname * aexp)\ set"$
and $"kill"$ :: $"com \Rightarrow (vname * aexp)\ set"$ **where**

definition AD :: $"(vname * aexp)\ set \Rightarrow com \Rightarrow (vname * aexp)\ set"$ **where**
$"AD\ A\ c = gen\ c \cup (A - kill\ c)"$

The defining equations for both gen and $kill$ follow the **where** and are separated by “|” as usual.

A call $AD\ A\ c$ 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 $"[\![\ (c,s) \Rightarrow s';\ \ \forall\ (x,a) \in A.\ s\ x = aval\ a\ s\ ]\!]$
$\Longrightarrow \forall\ (x,a) \in AD\ A\ c.\ s'\ x = aval\ a\ s'"$

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 $bury$ in the context of true liveness analysis (theory $Live\_True$) and prove that it is idempotent.

fun $bury$ :: $"com \Rightarrow vname\ set \Rightarrow com"$

The following two tweaks improve proof automation:

declare $L.simps(5)[simp]$
lemmas $L\_mono2 = L\_mono[unfolded\ mono\_def]$

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

lemma L_bury: $"X \subseteq Y \Longrightarrow L\ (bury\ c\ Y)\ X = L\ c\ X"$

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:

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:

lemma $"((W\ (\lambda s.\ s\ ''x'' \neq 0)\ (\{(s,t).\ t = s(''x'' := s\ ''x'' - 1)\}))\ \hat{}\hat{}n)\ \{\} =$
$\{(s,t).\ 0 \leqslant s\ ''x''\ \&\ s\ ''x'' < int\ n\ \&\ t = s(''x'' := 0)\}"$

Exercise 11.3. Consider Example 11.14 but with the loop condition $b = Less\ (N\ 0)\ (V\ ''x'')$. 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 $D\ (IF\ b\ THEN\ c1\ ELSE\ c2)$ in a point free way.

definition B :: $"bexp \Rightarrow (state \times state)\ set"$
lemma
$"D\ (IF\ b\ THEN\ c1\ ELSE\ c2) = (B\ b\ O\ D\ c1) \cup (B\ (Not\ b)\ O\ D\ c2)"$

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:

definition *thin* :: $"'a\ rel \Rightarrow\ 'a\ rel"$ **where**
$"thin\ R = \{(a,b)\ .\ (a,b) \in R \wedge (\forall\ c.\ (a,c) \in R \longrightarrow c = b)\}"$

Prove

lemma fixes f :: $"'a\ rel \Rightarrow\ 'a\ rel"$
assumes $"mono\ f"$ **and** *thin_f*: $"\bigwedge R.\ f\ (thin\ R) \subseteq thin\ (f\ R)"$
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 $\leqslant$ that have a least element $\bot$ and where every chain $c\ 0 \leqslant c\ 1 \leqslant \ldots$ has a least upper bound $lub\ c$ where $c :: nat \Rightarrow\ 'a$. This setting is described by the following type class *cpo* which is an extension of the type class *order* of partial orders. For details see the decription of type classes in Chapter 13.

context *order*
begin
definition *chain* :: $"(nat \Rightarrow\ 'a) \Rightarrow bool"$ **where**
$"chain\ c = (\forall n.\ c\ n \leqslant c\ (Suc\ n))"$
end

class *cpo* = *order* +
fixes *bot* :: *'a* **and** *lub* :: "$(nat \Rightarrow 'a) \Rightarrow 'a$"
assumes *bot_least*: "$bot \leqslant x$"
and *lub_ub*: "$chain\ c \Longrightarrow c\ n \leqslant lub\ c$"
and *lub_least*: "$chain\ c \Longrightarrow (\bigwedge n.\ c\ n \leqslant u) \Longrightarrow lub\ c \leqslant u$"

A function f :: $'a \Rightarrow 'b$ between two cpos $'a$ and $'b$ is called **continuous** if $f\ (lub\ c) = lub\ (f \circ c)$. Prove that if f is monotone and continuous then $lub\ (\lambda n.\ (f\ \hat{}\hat{}\ n)\ \bot)$ is the least (pre)fixpoint of f:

definition *cont* :: "$('a::cpo \Rightarrow 'b::cpo) \Rightarrow bool$" **where**
"$cont\ f = (\forall c.\ chain\ c \longrightarrow f\ (lub\ c) = lub\ (f\ o\ c))$"

abbreviation "$fix\ f \equiv lub\ (\lambda n.\ (f\hat{}\hat{}n)\ bot)$"

lemma *fix_lpfp*: **assumes** "*mono f*" **and** "$f\ p \leqslant p$" **shows** "$fix\ f \leqslant p$"
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 $(x,\ y) \in Dep\ c$ means that in the execution of c the initial value of x can influence the final value of y:

fun *Dep* :: "$com \Rightarrow (vname * vname)\ set$" **where**
"$Dep\ SKIP = Id$" |
"$Dep\ (x::=a) = \{(u,v).\ if\ v = x\ then\ u \in vars\ a\ else\ u = v\}$" |
"$Dep\ (c1;;c2) = Dep\ c1\ O\ Dep\ c2$" |
"$Dep\ (IF\ b\ THEN\ c1\ ELSE\ c2) = Dep\ c1 \cup Dep\ c2 \cup vars\ b \times UNIV$" |
"$Dep\ (WHILE\ b\ DO\ c) = lfp(\lambda R.\ Id \cup vars\ b \times UNIV \cup Dep\ c\ O\ R)$"

where $\times$ is the cross product of two sets. Prove monotonicity of the function *lfp* is applied to.

For the correctness statement define

abbreviation *Deps* :: "$com \Rightarrow vname\ set \Rightarrow vname\ set$" **where**
"$Deps\ c\ X \equiv (\bigcup x{\in}X.\ \{y.\ (y,x) : Dep\ c\})$"

and prove

lemma "$[\![\ (c,s) \Rightarrow s';\ \ (c,t) \Rightarrow t';\ \ s = t\ on\ Deps\ c\ X\ ]\!] \Longrightarrow s' = t'\ on\ X$"

Give an example that the following stronger termination-sensitive property

$[\![(c,\ s) \Rightarrow s';\ s = t\ on\ Deps\ c\ X]\!] \Longrightarrow \exists t'.\ (c,\ t) \Rightarrow t' \wedge s' = t'\ on\ X$

does not hold. Hint: $X = \{\}$.

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

$lvars\ c1 \cup lvars\ c2$ (in the case $IF\ b\ THEN\ c1\ ELSE\ c2$) and by $lvars\ c$ (in the case $WHILE\ b\ DO\ c$). Adjust the proof of the above correctness statement.

Chapter 12

Exercise 12.2. Define *bsubst* and prove the Substitution Lemma:

fun *bsubst* :: "*bexp* ⇒ *aexp* ⇒ *vname* ⇒ *bexp*"
lemma *bsubstitution*: "*bval* (*bsubst b a x*) *s* = *bval b* (*s*[*a*/*x*])"

This may require a similar definition and proof for *aexp*.

Exercise 12.3. Define a command *cmax* that stores the maximum of the values of the IMP variables x and y in the IMP variable z and prove that *cmax* satisfies its specification:

abbreviation *cmax* :: *com*
lemma "⊢ {λ*s*. *True*} *cmax* {λ*s*. *s* ''*z*'' = *max* (*s* ''*x*'') (*s* ''*y*'')}"

Function *max* is the predefined maximum function. Proofs about *max* are often automatic when simplifying with *max_def*.

Exercise 12.4. Define an equality operation for arithmetic expressions

definition *Eq* :: "*aexp* ⇒ *aexp* ⇒ *bexp*"

such that

lemma *bval_Eq*[*simp*]: "*bval* (*Eq a1 a2*) *s* = (*aval a1 s* = *aval a2 s*)"

Prove the following variant of the summation command correct:

```
lemma
  "⊢ {λs. s ''x'' = i ∧ 0 ⩽ i}
    ''y'' ::= N 0;;
    WHILE Not(Eq (V ''x'') (N 0))
    DO (''y'' ::= Plus (V ''y'') (V ''x'');;
         ''x'' ::= Plus (V ''x'') (N (−1)))
    {λs. s ''y'' = sum i}"
```

Exercise 12.5. Prove that the following command computes $y - x$ if $0 \leqslant x$:

```
lemma
  "⊢ {λs. s ''x'' = x ∧ s ''y'' = y ∧ 0 ⩽ x}
    WHILE Less (N 0) (V ''x'')
     DO (''x'' ::= Plus (V ''x'') (N (−1));; ''y'' ::= Plus (V ''y'') (N
(−1)))
    {λt. t ''y'' = y − x}"
```

Exercise 12.6. Define and verify a command *cmult* that stores the product of x and y in z assuming $0 \leqslant y$:

abbreviation *cmult* :: *com*
lemma
  "⊢ {λ*s*. *s* ''*x*'' = *x* ∧ *s* ''*y*'' = *y* ∧ 0 ⩽ *y*} *cmult* {λ*t*. *t* ''*z*'' = *x*∗*y*}"

You may have to simplify with *algebra_simps* to deal with "∗".

Exercise 12.7. The following command computes an integer approximation r of the square root of $i \geqslant 0$, i.e. $r^2 \leqslant i < (r+1)^2$. Prove

lemma
$$
\begin{array}{l}
\text{"}\vdash \{\ \lambda s.\ s\ ''x'' = i \wedge 0 \leqslant i\} \\
\quad ''r'' ::= N\ 0;;\ ''r2'' ::= N\ 1;; \\
\quad WHILE\ (Not\ (Less\ (V\ ''x'')\ (V\ ''r2''))) \\
\quad DO\ (''r'' ::= Plus\ (V\ ''r'')\ (N\ 1);; \\
\qquad\quad ''r2'' ::= Plus\ (V\ ''r2'')\ (Plus\ (Plus\ (V\ ''r'')\ (V\ ''r''))\ (N\ 1))) \\
\quad \{\lambda s.\ (s\ ''r'')\ \hat{}2 \leqslant i \wedge i < (s\ ''r'' + 1)\ \hat{}2\}\text{"}
\end{array}
$$

Figure out how $r2$ 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:

lemma $"\vdash \{P\}\ c\ \{\lambda s.\ True\}"$

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 $\exists$.

lemma $"\vdash \{P\}\ x ::= a\ \{Questionmark\}"$

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

Exercise 12.10. Prove

lemma $"\models \{P\}\ c\ \{Q\} \longleftrightarrow (\forall s.\ P\ s \longrightarrow wp\ c\ Q\ s)"$

Exercise 12.11. Replace the assignment command with a new command $Do\ f$ where $f :: state \Rightarrow state$ 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!

lemma $"[\![\vdash \{P\}\ c\ \{Q\};\ \vdash \{P'\}\ c\ \{Q'\}]\!] \Longrightarrow$
$\vdash \{\lambda s.\ P\ s \vee P'\ s\}\ c\ \{\lambda s.\ Q\ s \vee Q'\ s\}"$
lemma $"[\![\vdash \{P\}\ c\ \{Q\};\ \vdash \{P'\}\ c\ \{Q'\}]\!] \Longrightarrow$
$\vdash \{\lambda s.\ P\ s \wedge P'\ s\}\ c\ \{\lambda s.\ Q\ s \wedge Q'\ s\}"$
lemma $"[\![\vdash \{P\}\ c\ \{Q\};\ \vdash \{P'\}\ c\ \{Q'\}]\!] \Longrightarrow$
$\vdash \{\lambda s.\ P\ s \longrightarrow P'\ s\}\ c\ \{\lambda s.\ Q\ s \longrightarrow Q'\ s\}"$

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 *REPEAT* loop. Hint: think of *REPEAT c UNTIL b* as equivalent to *c;; 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 :: "$com \Rightarrow assn \Rightarrow assn$"

Prove that *sp* really is the strongest postcondition:

lemma "$(\models \{P\}\ c\ \{Q\}) \longleftrightarrow (\forall s.\ sp\ c\ P\ s \longrightarrow Q\ s)$"

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 $prevc$ :: "$acom \Rightarrow assn \Rightarrow assn \times bool$"

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 $post$ :: "$acom \Rightarrow assn \Rightarrow assn$"

such that (with the execption 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 is uses *post* instead of *pre* and prove its soundness and completeness.

fun vc :: "$acom \Rightarrow assn \Rightarrow bool$"
lemma vc_sound: "$vc\ C\ P \Longrightarrow\ \vdash \{P\}\ strip\ C\ \{post\ C\ P\}$"
lemma $vc\_complete$: "$\vdash \{P\}\ c\ \{Q\}$
$\Longrightarrow \exists C.\ strip\ C = c \wedge vc\ C\ P \wedge (\forall s.\ post\ C\ P\ s \longrightarrow Q\ s)$"

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 *acom* by annotating *WHILE* with a measure function in addition to an invariant:

Awhile assn "state ⇒ nat" bexp acom
("({_, _}/ *WHILE* _/ *DO* _)" [0, 0, 61] 61)

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

lemma *vc_sound*: "$vc\ C\ Q \Longrightarrow\ \vdash_t \{pre\ C\ Q\}\ strip\ C\ \{Q\}$"

You may need the combined soundness and completeness of $\vdash_t$: $(\vdash_t \{P\}\ c\ \{Q\}) = (\models_t \{P\}\ c\ \{Q\})$

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 *com*, then extend type *acom* with a corresponding construct:

```
Or "'a acom" "'a acom" 'a     ("_ OR// _//_"  [60, 61, 0] 60)
```

Finally extend function *Step*. Update proofs as well. Hint: think of *OR* as a nondeterministic conditional without a test.

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

lemma fixes $x0$ :: "$'a$ :: $order$"
assumes "$\bigwedge x\ y.\ x \leqslant y \Longrightarrow f\ x \leqslant f\ y$" **and** "$f\ q \leqslant q$" **and** "$x0 \leqslant q$"
shows "$(f$ ^^ $i)\ x0 \leqslant q$"

lemma fixes $x0$ :: "$'a$ :: $order$"
assumes "$\bigwedge x\ y.\ x \leqslant y \Longrightarrow f\ x \leqslant f\ y$" **and** "$x0 \leqslant f\ x0$"
shows "$(f$ ^^ $i)\ x0 \leqslant (f$ ^^ $(i{+}1))\ x0$"

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

lemma fixes P :: "$'a$::$complete\_lattice\ set$"
assumes "$mono\ f$" **and** "$\forall p \in P.\ f\ p \leqslant p$"
shows "$f(\bigsqcap\ P) \leqslant \bigsqcap\ P$"

Sledgehammer should give you a proof you can start from.

Exercise 13.16. Give a readable proof that if $\gamma :: 'a{::}lattice \Rightarrow 'b{::}lattice$ is a monotone function, then $\gamma\ (a_1 \sqcap a_2) \leqslant \gamma\ a_1 \sqcap \gamma\ a_2$:

lemma fixes γ :: "$'a$::$lattice \Rightarrow 'b$ :: $lattice$"
assumes $mono$: "$\bigwedge x\ y.\ x \leqslant y \Longrightarrow \gamma\ x \leqslant \gamma\ y$"
shows "$\gamma\ (a_1 \sqcap a_2) \leqslant \gamma\ a_1 \sqcap \gamma\ a_2$"

Give an example of two lattices and a monotone γ where $\gamma\ a_1 \sqcap \gamma\ a_2 \leqslant \gamma\ (a_1 \sqcap a_2)$ does not hold.

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

datatype $sign$ = $None$ | Neg | $Pos0$ | Any

fun γ :: "$sign \Rightarrow val\ set$" **where**
"$\gamma\ None = \{\}$" |
"$\gamma\ Neg = \{i.\ i < 0\}$" |

$"\gamma\ Pos0 = \{i.\ i \geqslant 0\}"$ |
$"\gamma\ Any = UNIV"$

Define inverse analyses for "+" and "<" and prove the required correctness properties:

fun *inv_plus'* :: "*sign* ⇒ *sign* ⇒ *sign* ⇒ *sign* ∗ *sign*"
lemma
$"\llbracket\ inv\_plus'\ a\ a1\ a2 = (a1',a2');\ \ i1 \in \gamma\ a1;\ \ i2 \in \gamma\ a2;\ i1{+}i2 \in \gamma\ a\ \rrbracket$
$\Longrightarrow i1 \in \gamma\ a1' \wedge i2 \in \gamma\ a2'\ "$
fun *inv_less'* :: "*bool* ⇒ *sign* ⇒ *sign* ⇒ *sign* ∗ *sign*"
lemma
$"\llbracket\ inv\_less'\ bv\ a1\ a2 = (a1',a2');\ \ i1 \in \gamma\ a1;\ \ i2 \in \gamma\ a2;\ (i1{<}i2) = bv\ \rrbracket$
$\Longrightarrow i1 \in \gamma\ a1' \wedge i2 \in \gamma\ a2'"$

For the ambitious: turn the above fragment into a full-blown abstract interpreter by replacing the interval analysis in theory *Abs_Int2_ivl* by a sign analysis.