Concrete Semantics with Isabelle/HOL
Exercise Sheet 3

Exercise 3.1 Relational aval

Theory $AExp$ defines an evaluation function $aval ::= aexp \Rightarrow state \Rightarrow val$ for arithmetic expressions. Define a corresponding evaluation relation $is\_aval ::= aexp \Rightarrow state \Rightarrow val \Rightarrow bool$ as an inductive predicate:

\[
\text{inductive } is\_aval :: \text{"aexp \Rightarrow state \Rightarrow val \Rightarrow bool\"}
\]

Use the introduction rules $is\_aval\text{.intros}$ to prove this example lemma.

\[
\text{lemma } \text{"is\_aval (Plus (N 2) (Plus (V x) (N 3))) s (2 + (s x + 3))"}
\]

Prove that the evaluation relation $is\_aval$ agrees with the evaluation function $aval$. Show implications in both directions, and then prove the if-and-only-if form.

\[
\text{lemma } aval1: \text{"is\_aval a s v \implies aval a s = v"}
\]
\[
\text{lemma } aval2: \text{"aval a s = v \implies is\_aval a s v"}
\]
\[
\text{theorem } \text{"is\_aval a s v \iff aval a s = v"}
\]

Exercise 3.2 Avoiding Stack Underflow

A stack underflow occurs when executing an instruction on a stack containing too few values—e.g., executing an $ADD$ instruction on an stack of size less than two. A well-formed sequence of instructions (e.g., one generated by $comp$) should never cause a stack underflow.

In this exercise, you will define a semantics for the stack machine that throws an exception if the program underflows the stack.

Modify the $exec\_1$ and $exec$ functions so that they return an option value, $None$ indicating a stack underflow.

\[
\text{fun } exec\_1 :: \text{"instr \Rightarrow state \Rightarrow stack \Rightarrow stack option\"}
\]
\[
\text{fun } exec :: \text{"instr list \Rightarrow state \Rightarrow stack \Rightarrow stack option\"}
\]

Adjust the proof of theorem $exec\_comp$ to show that programs output by the compiler never underflow the stack:

\[
\text{theorem } exec\_comp: \text{"exec (comp a) s stk = Some (aval a s \# stk)"}
\]
**Exercise 3.3** Boolean If Expressions

We consider an alternative definition of Boolean expressions featuring a conditional construct:

```plaintext
datatype ifexp = Bc' bool | If ifexp ifexp ifexp | Less' aexp aexp
```

1. Define a function `ifval` analogous to `bval`, which evaluates `ifexp` expressions.
2. Define a function `translate`, which translates `ifexp`s to `bexp`s. State and prove a lemma showing that the translation is correct.

**Exercise 3.4** A Bit of Isar

```plaintext
fun sum_n :: "nat ⇒ nat" where
  "sum_n 0 = 0" |
  "sum_n (Suc n) = Suc n + sum_n n"
```

In homework 1.2, we formalized a function summing the first $n + 1$ natural numbers. We proved $\sum_n n = n \cdot (n + 1) \div 2$ both informally and using an `apply` script. Take your informal proof and transform it into a structured Isar proof, respecting its structure as much as possible.

Hint: The `also` and `finally` Isar keywords might come in handy.

```
lemma "sum_n n = n \cdot (n + 1) \div 2"
```

**Homework 3.1** Boolean Expression (12 points)

Submission until Monday 18.05.2015 10:00 AM. Please send your submissions to jasmin.blanchette@mpi-inf.mpg.de and mathias.fleury@mpi-inf.mpg.de with “[CONCRETE3]” in the subject of your email. If your name is Vorname Nachname, please call your theory file `Nachname_Vorname.thy`.

This homework is about compiling Boolean expressions to a state machine. This state machine has the following instructions:

**Step A (The State Machine, 4 points)**

```plaintext
type_synonym stack = "val list"
abbreviation "hd2 xs == hd (tl xs)"
abbreviation "tl2 xs == tl (tl xs)"
abbreviation "intOfBool ≡ (λs. if s then 1 else 0)"

datatype instr =
```

2
LESS — compare two values
| AND — conjunction
| NOT — negation
| JMPF nat — forward jump by offset if the head of the stack contains false
| ADD — addition
| LOAD varname — value of a variable
| LOADI val — loads a constant

We consider that 0 is false and every other value is true (like in C).

Define the execution of a single instruction. Note that JMPF n has no effect in itself.

fun exec₁ :: "instr ⇒ state ⇒ stack ⇒ stack" where
"exec₁ (JMPF n) stk = stk" |

Now define the execution of a list of instructions. You might need drop n that drops the first n elements of a list to implement JMPF.

fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack"

Prove or disprove:

lemma exec_append: “exec (is₁ @ is₂) s stk = exec is₂ s (exec is₁ s stk)"

Step B (Compilation, 4 points)

Define the compilation of arithmetic expressions:

fun acomp :: “aexp ⇒ instr list”

Define the compilation of Boolean expressions:

fun bcomp :: “bexp ⇒ instr list”

Prove

lemma exec_acomp[simp]: “exec (acomp a) s stk = aval a s # stk”

lemma exec_bcomp[simp]: “exec (bcomp a) s stk = (intOfBool (bval a s)) # stk”

Hint: You will need to prove a lemma like exec_append or perhaps exec_append itself.

Step C (Shortcuts, 4 points)

Now compile the expression with shortcuts. Since the test in LOADI 0 # bcomp x @ [AND] is always false (irrespective of the value of x), we can skip the second evaluation (with a JMPF). Do not change the evaluation function, only the compiler.

The jump should only be to the end of the corresponding AND: in α ≠ β @ [AND] @ γ @ [AND], there will a jump from α to the start of γ and from γ to after the second AND. An optimization would to jump directly from α to the second AND.
fun bcomp' :: "bexp ⇒ instr list"

Prove that the execution of the code generated by the two compilers has the same semantics:

**lemma** exec_bcomp_bcomp: "exec (bcomp' a) s stk = exec (bcomp a) s stk"

**Step D (More Boolean Expressions, optional, 0 points)**

Create the following expression either on *bexp* level or on *instr list*, based on *bexp.Not*, *And*, *Or* when you have defined it, and so one. *Ite* means if–then–else.

**definition** Or :: "bexp ⇒ bexp ⇒ bexp"
**definition** Imp :: "bexp ⇒ bexp ⇒ bexp"
**definition** Ite :: "bexp ⇒ bexp ⇒ bexp ⇒ bexp"
**definition** OR :: "instr list ⇒ instr list ⇒ instr list"
**definition** IMP :: "instr list ⇒ instr list ⇒ instr list"
**definition** ITE :: "instr list ⇒ instr list ⇒ instr list ⇒ instr list"

Prove the correctness of *Or*, *Imp*, and *Ite*:

**lemma** exec_Or:
  "exec (OR (bcomp' e_1) (bcomp' e_2) @ l) s stk =
  exec l s (intOfBool (bval e_1 s ∨ bval e_2 s) # stk)"

The next two lemmas are not difficult, but you will have to play with the parameters of *simp*. Read carefully you goal. You will want to deactivate some *simp* rules with *del*: and add some *simp* rules with *add:*

The simplification facts of a function like *bcomp* are called *bcomp.simps*. You might also need to write *bcomp.simps[ symmetric]* to reorient the rules.

**lemma** exec_Imp:
  "exec (IMP (bcomp' e_1) (bcomp' e_2) @ l) s stk =
  exec l s (intOfBool (bval e_1 s −→ bval e_2 s) # stk)"

**lemma** exec_ITE:
  "exec (ITE (bcomp' e_1) (bcomp' e_2) (bcomp' e_3) @ l) s stk =
  exec l s (intOfBool (if bval e_1 s then bval e_2 s else bval e_3 s) # stk)"

**Homework 3.2** Easy as ABC (8 points)

Submission until Monday 18.05.2015 10:00 AM. Please send your submissions to jasmin.blanchette@mpi-inf.mpg.de and mathias.fleury@mpi-inf.mpg.de with “[CONCRETE3]” in the subject of your email. If your name is Vorname Nachname, please call your theory file Nachname_Vorname.thy.
We will work towards defining an inductive set specifying the language consisting of words of the form \(a^m b^n c^m\). First, let us define the alphabet:

\[
\text{datatype } \Sigma = A \mid B \mid C
\]

\[
\text{type synonym } \text{word} = "\Sigma \ list"
\]

In keeping with Isabelle conventions, we use uppercase names for constructors.

**Step A (6 points):** Towards the above goal, it helps to start by defining the language of words of the form \(b^n\). Here is how to do it inductively:

\[
\text{inductive set } B \text{words} :: "word set" \text{ where}
\]

\[
B \text{words}\ Nil[\text{intro}]: "\[] \in B \text{words}"
\]

\[
B \text{words}\ Cons[\text{intro}]: "w \in B \text{words} \Rightarrow B \# w \in B \text{words}"
\]

We register the introduction rules as \text{intro} rules so that they are automatically applied by \text{auto}, \text{blast}, \text{flabbergast}, and friends. These rules work well because they terminate (the inductive predicate is well-founded)—the assumption of the second rule is on a shorter word than the conclusion.

An alternative way of characterizing those words relies on the \text{replicate} function from the standard list library:

\[
\text{definition } B \text{of} :: "nat} \Rightarrow \text{word" where}
\]

\[
"B \text{of} n = \text{replicate} n B"
\]

Prove that the two characterizations are equivalent. (If you use Isar, \text{case} and \text{print_cases} might be useful.)

\[
\text{lemma } B \text{words\ sound}: "w \in B \text{words} \Rightarrow \exists n. w = B \text{of} n"
\]

\[
\text{lemma } B \text{words\ complete}: "\exists n. w = B \text{of} n \Rightarrow w \in B \text{words}"
\]

**Step B (2 points):** Building on \(B \text{words}\), define an inductive set called \(ABC\text{words}\) that captures all words of the form \(a^m b^n c^m\):

\[
\text{inductive set } ABC \text{words} :: "word set"
\]

Define \(ABC\text{of}\) analogously to \(B\text{of}\), but this time taking both \(m\) and \(n\) as arguments:

\[
\text{definition } ABC \text{of} :: "nat} \Rightarrow nat} \Rightarrow \text{word"}
\]

Finally, prove the equivalence of both characterizations.

\[
\text{lemma } ABC \text{words\ sound}: "w \in ABC \text{words} \Rightarrow \exists m. n. w = ABC \text{of} m n"
\]

\[
\text{lemma } ABC \text{words\ complete}: "\exists m. n. w = ABC \text{of} m n \Rightarrow w \in ABC \text{words}"
\]

**Homework 3.3 A Bit² of Isar (5 bonus points)**

*Submission until Monday 18.05.2015 10:00 AM. Please send your submissions to jasmin.blanchette@mpi-inf.mpg.de and mathias.fleury@mpi-inf.mpg.de with “[CONCRETE3]”*
in the subject of your email. If your name is Vorname Nachname, please call your theory file Nachname_Vorname.thy.

The sum of the squares of the natural numbers from 1 to \( n \) is given by the formula
\[
\sum_{i=1}^{n} i^2 = \frac{n(n + 1)(2n + 1)}{6}.
\]
Show this via a structured Isar proof, either using the built-in summation of Isabelle (defined in Main) or by defining your own “sum square” function.

For 0 additional bonus points, try to do an apply-style proof. What do you observe?