Goal You can use inductively defined relations to describe the semantic of programs and perform simple proofs over these semantics. Also, you get to know the concept of history variables.

Note The basic definitions can be downloaded from the homepage.

Exercise 1 [10] Execution of machine code

Consider the following simple assembly language. All operations work on a single register. Set, Add, and Mul perform simple arithmetic with constants. Load and Store access a memory of named values.

datatype instr =
  Set int | Add int | Mul int |
  Load string | Store string

A block of instructions and the state can be easily defined.

type-synonym vals = "string ↦ int"

type-synonym block = "instr list"

record state =
  vals :: vals
  accu :: int
  err :: bool

Here we see an alternative to the Some/None approach to error handling. The field err in the state is a history variable. At the beginning this variable is False and it will be set to True if an error occurs. So, if an error occurs this variable is True at the end of execution. We do not need to do a case analysis in every step.

(a) [5] Define the execution of a block by an inductively defined relation. To make proofs about this relation easier make sure that the result state in each rule is a variable.

  inductive exec :: "state ⇒ block ⇒ state ⇒ bool"

An error occurs if a variable undefined in the state is read or written. Use the helper function

  fun mark-err :: "state ⇒ state"
  where "mark-err s = (s (| err := True |))"

Test your definition by proving the following lemmas:

  exec (init ["x" ↦ 3, "y" ↦ 4] 0) [ Load "x", Add 3 ] (| vals = ["x" ↦ 3, "y" ↦ 4], accu= 6, err = False |)
  exec (init ["x" ↦ 3, "y" ↦ 4] 0) [ Load "x", Mul 2, Store "y", Set 0 ] (| vals = ["x" ↦ 3, "y" ↦ 6], accu=0, err= False |)
  exec (init ["x" ↦ 3, "y" ↦ 4] 0) [ Load "z", Mul 2 ] (| vals = ["x" ↦ 3, "y" ↦ 4], accu=0, err = True |)

(b) [4] Define a function

  block-vars :: "block ⇒ string set"

which computes the list of all variables accessed by a block. Prove then that no error occurs if all accessed variables are defined.

  [ exec s b s'; block-vars b ⊆ dom (vals s); ¬ err s ] ⇒ ¬ err s'}
Prove this lemma

- by induction over the execution with the rule exec.induct
- and by induction over the structure of a block

For the latter case, you get \texttt{exec s b s'} as a premise and you need to do a case distinction on the execution predicate (which was done by the induction rule previously). You can use the rule exec.cases for that.

Also, the \texttt{try0} and \texttt{try} commands might prove useful. They invoke a list of common methods to find a proof. \texttt{try0} also accepts the usual parameters like \texttt{simp:} and \texttt{intro:}.

Many methods (for example \texttt{auto}, \texttt{simp}, and \texttt{fastforce} also take a \texttt{split:} parameter to automatically perform splits on case-expressions. The split rules for the option datatype are called \texttt{option.split} and \texttt{option.split_asm}.

(c) [1] Prove that the opposite of this lemma also holds, i.e., if the execution terminates without error, all accessed variables must be defined in the initial state.

\[
\lfloor \text{exec } s \ b \ s' ; \neg \text{err } s' \rfloor \implies \text{block-vars } b \subseteq \text{dom (vals s)}
\]

(d) [3 (opt.)] Prove that the execution always terminates.

\[
\forall s. \exists s'. \text{exec } s \ b \ s'
\]