Isabelle/HOL Exercises
Advanced

Compiler for Register Machine from Hell

Processors from Hell has released its next-generation RISC processor. It features an infinite bank of registers R_0, R_1, etc, holding unbounded integers. Register R_0 plays the role of the accumulator and is the implicit source or destination register of all instructions. Any other register involved in an instruction must be distinct from R_0. To enforce this requirement the processor implicitly increments the index of the other register. There are 4 instructions:

`LDI` i has the effect $R_0 := i$

`LD` n has the effect $R_0 := R_{n+1}$

`ST` n has the effect $R_{n+1} := R_0$

`ADD` n has the effect $R_0 := R_0 + R_{n+1}$

where i is an integer and n a natural number.

Define a datatype of instructions

datatype *instr*

and an execution function that takes an instruction and a state

type_synonym *state = "nat ⇒ int"*

and returns the new state:

```
exec :: "instr ⇒ state ⇒ state"
```

Extend *exec* to instruction lists:

```
execs :: "instr list ⇒ state ⇒ state"
```

A source language

The engineers at *PfH* soon got tired of writing assembly language code and designed their own high-level programming language of arithmetic expressions. An expression can be

- an integer constant,

- one of the variables V_0, V_1, ..., or
- the sum of two expressions.

Define a datatype of expressions

datatype *expr*

and an evaluation function that takes an expression and a state and returns the value:

```
val :: "expr ⇒ state ⇒ int"
```

Because this is a clean language, there is no implicit incrementation going on: the value of V_n in state s is simply $s(n)$.

A compiler

You have been recruited to write a compiler from *expr* to *instr list*. You remember your compiler course and decide to emulate a stack machine using free registers, i.e. registers not used by the expression you are compiling. The type of your compiler is

```
cmp :: "expr ⇒ nat ⇒ instr list"
```

where the second argument is the index of the first free register and can be used to store intermediate results. The result of an expression should be returned in R_0. Because R_0 is the accumulator, you decide on the following compilation scheme: V_i will be held in R_{i+1}.

Having earned a PhD in theoretical computer science you want to impress your boss and colleagues at *PfH* by verifying your compiler. Unfortunately your colleagues could not care less, and your boss explicitly forbids you to pursue this ill-guided project during working hours. As a result you decide to take a week's holiday in the Austrian Alps to work hard on your proof. On the train you have already sketched the following correctness statement:

```
execs (cmp e k) s 0 = val e s
```

However, there is definitely a precondition missing because k should be large enough not to interfere with any of the variables in `e`. Moreover, you have some lingering doubts about having the same `s` on both sides despite the index shift between variables and registers. But because all your definitions are executable, you hope that Isabelle will spot any incorrect proposition before you even start its proof. What worries you most is the number of auxiliary lemmas it may take to prove your proposition.