Isabelle/HOL Exercises
Projects

Optimising Compilation for a Register Machine

Section 3.3 of the Isabelle/HOL tutorial describes an expression compiler for a stack machine. In this exercise we will build and verify an optimising expression compiler for a register machine.

The Source Language: Expressions

The arithmetic expressions we will work with consist of variables, constants, and an arbitrary binary operator `oper`.

```
consts oper :: "nat ⇒ nat ⇒ nat"

type_synonym var = string

datatype exp =
    Const nat
  | Var var
  | Op exp exp
```

The state in which an expression is evaluated is modelled by an *environment* function that maps variables to constants.

```
type_synonym env = "var ⇒ nat"
```

Define a function `value` that evaluates an expression in a given environment.

```
consts "value" :: "exp ⇒ env ⇒ nat"
```

The Register Machine

As the name suggests, a register machine uses a collection of registers to store intermediate results. There exists a special register, called the accumulator, that serves as an implicit argument to each instruction. The rest of the registers make up the register file, and can be randomly accessed using an index.

```
type_synonym regIndex = nat

datatype cell =
    Acc
  | Reg regIndex
```

The state of the register machine is denoted by a function that maps storage cells to constants.

```
type_synonym state = "cell ⇒ nat"
```

The instruction set for the register machine is defined as follows:

```
datatype instr =
  LI nat
  — Load Immediate: loads a constant into the accumulator.
| LOAD regIndex
  — Loads the contents of a register into the accumulator.
| STORE regIndex
  — Saves the contents of the accumulator in a register.
| OPER regIndex
  — Performs the binary operation oper.
    — The first argument is taken from a register.
    — The second argument is taken from the accumulator.
    — The result of the computation is stored in the accumulator.
```

A program is a list of such instructions. The result of running a program is a change of state of the register machine. Define a function `exec` that models this.

```
consts exec :: "state ⇒ instr list ⇒ state"
```

Compilation

The task now is to translate an expression into a sequence of instructions that computes it. At the end of execution, the result should be stored in the accumulator.

Before execution, the values of each variable need to be stored somewhere in the register file. A *mapping* function maps variables to positions in the register file.

```
type_synonym map = "var ⇒ regIndex"
```

Define a function `cmp` that compiles an expression into a sequence of instructions. The evaluation should proceed in a bottom-up depth-first manner.

State and prove a theorem expressing the correctness of `cmp`.

Hints:

- The compilation function is dependent on the mapping function.

- The compilation function needs some way of storing intermediate results. It should be clever enough to reuse registers it no longer needs.
- It may be helpful to assume that at each recursive call, compilation is only allowed to use registers with indices greater than a given value to store intermediate results.

Compiler Optimisation: Common Subexpressions

In the previous section, the compiler `cmp` was allowed to evaluate a subexpression every time it occurred. In situations where arithmetic operations are costly, one may want to compute commonly occurring subexpressions only once.

For example, to compute `(a op b) op (a op b)`, `cmp` was allowed three calls to `oper`, when only two were needed.

Develop an optimised compiler `optCmp`, that evaluates every commonly occurring subexpression only once. Prove its correctness.