Isabelle/HOL Exercises
Projects

Optimising Compilation 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"

lemma operI:"⟦a = c ; b = d⟧ ⟹ oper a b = oper c d"
by simp

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"
```

The function *value* evaluates an expression in a given environment.

```
primrec "value" :: "exp ⇒ env ⇒ nat" where
  "value (Const n) e = n"
| "value (Var v) e = (e v)"
| "value (Op e1 e2) e = (oper (value e1 e) (value e2 e))"
```

The Register Machine

Register indices and storage cells:

```
type_synonym regIndex = nat
```

datatype *cell* =
    Acc
  | *Reg regIndex*

The instruction set:

datatype *instr* =
    LI nat
  | *LOAD regIndex*
  | *STORE regIndex*
  | *OPER regIndex*

type_synonym *state* = "*cell* ⇒ *nat*"

Result of executing a single machine instruction:

```
primrec execi :: "state ⇒ instr ⇒ state" where
  "execi s (LI n) = (s (Acc := n))"
| "execi s (LOAD r) = (s (Acc := s(Reg r)))"
| "execi s (STORE r) = (s ((Reg r) := (s Acc)))"
| "execi s (OPER r) = (s (Acc := (oper (s (Reg r)) (s Acc))))"
```

Result of serially executing a sequence of machine instructions:

```
definition exec :: "state ⇒ instr list ⇒ state" where
  "exec s instrs == foldl execi s instrs"
```

Some lemmas about *exec*:

```
lemma exec_app:"exec s (p1 @ p2) = exec (exec s p1) p2"
by (clarsimp simp:exec_def)

lemma exec_null:"exec s [] = s"
by (clarsimp simp:exec_def)

lemma exec_cons:"exec s (i#is) = exec (execi s i) is"
by (clarsimp simp:exec_def)

lemma exec_sing:"exec s [i] = execi s i"
by (clarsimp simp:exec_def)
```

1 Compilation

A *mapping* function maps variables to positions in the register file.

type_synonym *map* = "*var* ⇒ *regIndex*"

The function *cmp* recursively translates an expression into a sequence of instructions that computes it. At the end of execution, the result is stored in the accumulator. In addition to a *mapping* function, *cmp* takes the next free register index as input.

```
primrec cmp :: "exp ⇒ map ⇒ regIndex ⇒ instr list" where
  "cmp (Const n) m r = [LI n]"
| "cmp (Var v) m r = [(LOAD (m v))]"
| "cmp (Op e1 e2) m r = (cmp e1 m r)@[STORE r]@
                          (cmp e2 m (Suc r))@[OPER r]"
```

The correctness lemma for *cmp*:

```
theorem corr_and_no_se:
  "⋀ st r.
       ⟦∀v. m v < r; ∀v. (env v) = (st (Reg (m v))) ⟧ ⟹
       ((exec st (cmp e m r)) Acc = value e env) ∧
       (∀x. (x < r) ⟶ (((exec st (cmp e m r)) (Reg x)) = st (Reg x)))"
  (is "⋀ st r. ⟦?vars_below_r st r ;?var_vals st r ⟧ ⟹
                                  ?corr st e r ∧ ?no_side_eff st e r")
```

Note:

- We need some way of giving *cmp* the start index *r* of the free region in the register file. Initially, *r* should be above all variable mappings. The first assumption ensures this.
- All variable mappings should agree with the *environment* used in *value*. The second assumption ensures this.
- The first part of the conclusion expresses the correctness of *cmp*.
- The second part of the conclusion expresses the fact that compilation does not use already allocated registers (i.e. those below *r*). This is needed for the inductive proof to go through.

```
proof (induct e)
  case Const show ?case by (clarsimp simp:exec_def)
next
  case Var assume "?var_vals st r" then show ?case by (clarsimp
simp:exec_def)
next
  case (Op e1 e2)
  assume hyp1:"⋀ st r. ⟦?vars_below_r st r ;?var_vals st r ⟧ ⟹
                        ?corr st e1 r ∧ ?no_side_eff st e1 r"
    and hyp2:"⋀ st r. ⟦?vars_below_r st r ;?var_vals st r ⟧ ⟹
                        ?corr st e2 r ∧ ?no_side_eff st e2 r"
    and vars_below_r:"?vars_below_r st r" and var_vals:"?var_vals st r"
```

— Four lemmas useful for simplification of main subgoal

```
have rw1:"⋀st x. ⟦ x < (Suc r); ?var_vals st r ⟧ ⟹
                    exec st (cmp e2 m (Suc r)) (Reg x) = st (Reg x)"
proof -
  fix st x
  assume a:"x < Suc r" and b:"?var_vals st r"
  from vars_below_r have "?vars_below_r st (Suc r)"
    apply clarify
    by (erule_tac x=v in allE, simp)
  with a b hyp2[of "(Suc r)" "st"]  show "?thesis st x" by clarsimp
qed

have rw2:"⋀st.?var_vals st r ⟹
          exec st (cmp e2 m (Suc r)) Acc = value e2 env"
proof -
  fix st
  assume a:"?var_vals st r"
  from vars_below_r have b: "?vars_below_r st (Suc r)"
    apply clarify
    by (erule_tac x=v in allE, simp)
  from b a show "?thesis st"
    by (rule hyp2[THEN conjunct1,of "(Suc r)" "st"])
qed

have rw3:"⋀st x. ⟦ x < r ; ?var_vals st r ⟧ ⟹
          exec st (cmp e1 m r) (Reg x) = st (Reg x)"
proof -
  fix st x
  assume a:"x < r" and b:"?var_vals st r"
  with vars_below_r hyp1[of "r" "st"]  show "?thesis st x" by clarsimp
qed

from vars_below_r var_vals
have val_e1:"exec st (cmp e1 m r) Acc = value e1 env"
  by (rule hyp1[THEN conjunct1,of "r" "st"])
```

— Two lemmas that express *?var_vals* also holds for the two states
— encountered in the proof of the main subgoal

```
from vars_below_r var_vals
have vB1:"?var_vals ((exec st (cmp e1 m r))
          (Reg r := exec st (cmp e1 m r) Acc)) r"
```

```
    by (auto simp:rw3)

  from vars_below_r var_vals
  have vB2:"?var_vals ((exec st (cmp e1 m r))(Reg r := value e1 env)) r"
    by (auto simp:rw3)

  show ?case
  proof
    show "?corr st (Op e1 e2) r"
      apply (simp add:exec_app exec_cons exec_null)
      apply (rule operI)
       apply (simp add:rw1 vB1 val_e1)
       apply (simp add:rw1 val_e1)
      apply (simp add:rw2 vB2)
      done
  next
    show "?no_side_eff st (Op e1 e2) r"
      apply clarify
      apply (simp add:exec_app exec_cons exec_null)
      apply (simp add:rw1 vB1)
      apply (simp add:rw3 var_vals)
      done
  qed
qed
```

2 Compiler Optimisation: Common Subexpressions

The optimised compiler `optCmp`, should evaluate every commonly occurring subexpression only once.

General idea:

- Generate a list of all sub-expressions occurring in a given expression. A given sub-expression in this list can only be 'one step' dependent on sub-expressions occurring earlier in the list. For example a possible list of sub-expressions for `(a op b) op (a op b)` is `[a,b,a op b,a,b,a op b,(a op b) op (a op b)]`.

- Note that the resulting sub-expression list specifies an order of evaluation for the given expression. The list in the above example is an evaluation sequence `cmp` would use. Since it contains duplicates, it is not what we want.

- Remove all duplicates from this list, in such a way, so as not to break the sub-expression list property (i.e. in case of a duplicate, remove the later occurance). For our example, this would result in *[a,b,a op b,(a op b) op (a op b)]*.

- Evaluate all expressions in this list in the order that they occur. Store previous results somewhere in the register file and use them to evaluate later sub-expressions.

The previous *mapping* function is extended to include all expressions, not just variables.

```
type_synonym expMap = "exp ⇒ regIndex"
```

Instead of a single expression, the new compilation function takes as input a list of expressions. It is assumed that this list satisfies the sub-expression property discussed above.

At each step, it will compute the value of an expression, store it in the register file, and update the *mapping* function to reflect this.

```
primrec optCmp :: "exp list ⇒ expMap ⇒ regIndex ⇒ instr list" where
  "optCmp [] m r = []"
| "optCmp (x#xs) m r = (case x of
    (Const n) ⇒
      [LI n]@[STORE r]@                       (optCmp xs (m(x := r)) (Suc r)) |
    (Var v) ⇒
      [(LOAD (m (Var v)))]@[STORE r]@         (optCmp xs (m(x := r)) (Suc r)) |
    (Op e1 e2) ⇒
      [LOAD (m e2)]@[OPER (m e1)]@[STORE r]@ (optCmp xs (m(x := r)) (Suc r))
  )"
```

The function `alloc` returns the register allocation done by *optCmp*:

```
primrec alloc :: "expMap ⇒ exp list ⇒ regIndex ⇒ expMap" where
  "alloc m [] r = m"
| "alloc m (e#es) r = alloc (m(e := r)) es (Suc r)"
```

Some lemmas about `alloc` and *optCmp*:

```
lemma allocApp:"⋀ m r. alloc m (as @ bs) r =
                alloc (alloc m as r) bs (r + length as)"
by (induct as, auto)

lemma allocNotIn:"⋀m r. e ∉ set es ⟹ alloc m es r e = m e"
by (induct es, auto)
```

Sequential search in a list:

```
primrec search :: "'a ⇒ 'a list ⇒ nat" where
  "search a [] = 0"
| "search a (x#xs) = (if (x=a) then 0 else Suc (search a xs))"
```

```
lemma searchLessLength:"⋀ a. a:set xs ⟹ search a xs < length xs"
by (induct xs, auto)

lemma allocIn:"⋀m r. ⟦distinct es; e ∈ set es⟧ ⟹
                (alloc m es r e) = r + search e es"
apply (induct es)
 apply (auto)
apply (frule_tac m="(m(e := r))" and r="(Suc r)" in allocNotIn)
apply simp
done

lemma optCmpApp:"⋀ i m r. i = length as ⟹
  optCmp (as@bs) m r = (optCmp as m r) @ (optCmp bs (alloc m as r) (r+i))"
apply (induct as)
 apply clarsimp
apply (case_tac a, auto)
done
```

The function *supExp* expresses the converse of the sub-expression property discussed earlier:

```
primrec supExp :: "exp list ⇒ bool" where
  "supExp [] = True"
| "supExp (e#es) = (case e of
                    (Const n) ⇒ supExp es |
                    (Var v) ⇒ supExp es  |
                    (Op e1 e2) ⇒ (supExp es) ∧ (e1 : set es) ∧ (e2 : set es)
  )"
```

A definition of *subExp* using *supExp* (a direct definition is harder!):

```
definition subExp :: "exp list ⇒ bool" where
  "subExp es == supExp (rev es)"
```

The correctness theorem for *optCmp*:

```
theorem opt_corr_and_no_se:
  "⋀ st r.
        ⟦∀e. (m e) < r; ∀v. (env v) = (st (Reg (m (Var v))));
                                    subExp es; distinct es ⟧ ⟹
        (∀e∈set es. (exec st (optCmp es m r)) (Reg ((alloc m es r) e))
        = value e env) ∧
        (∀x. (x < r) ⟶ (((exec st (optCmp es m r)) (Reg x)) = st (Reg x)))"
```

Note:

- As input, we have an arbitrary expression list that satisfies the sub-expression property.

- The assumption that this list is unique is not strictly required, but makes the proof easier.
- The rest of theorem bears resemblance to that of *cmp*.

```
apply (induct es rule:rev_induct)
 apply (clarsimp simp: exec_cons exec_null)
apply (simp only:optCmpApp exec_app subExp_def)
apply (case_tac x)

  — Const case
  apply clarsimp
  apply rule
   apply (simp add:allocApp)
   apply (simp add:exec_cons exec_null)
  apply rule
   apply clarsimp
   apply (simp add:allocApp)
   apply rule
    apply (simp add:exec_cons exec_null)
   apply (simp add:exec_cons exec_null)
   apply clarsimp
   apply (frule_tac m="m" and e="e" and r="r" in allocIn)
    apply assumption
   apply (frule searchLessLength)
   apply simp
  apply clarsimp
  apply (simp add:exec_cons exec_null)

  — Var case
 apply clarsimp
 apply rule
  apply (simp add:allocApp)
  apply (simp add:exec_cons exec_null)
  apply (frule_tac m="m" and r="r" in allocNotIn)
  apply clarsimp
 apply (simp add:allocApp)
 apply rule
  apply clarsimp
  apply rule
   apply (clarsimp simp add:exec_cons exec_null)
  apply (clarsimp simp add:exec_cons exec_null)
  apply (frule_tac m="m" and e="e" and r="r" in allocIn)
   apply assumption
  apply (frule searchLessLength)
```

```
  apply simp
 apply clarsimp
 apply (simp add:exec_cons exec_null)
```

— Op case

```
apply clarsimp
apply rule
 apply (simp add:allocApp)
 apply (simp add:exec_cons exec_null)
apply (simp add:allocApp)
apply rule
 apply clarsimp
 apply rule
 apply (clarsimp simp add:exec_cons exec_null)
 apply (clarsimp simp add:exec_cons exec_null)
 apply (frule_tac m="m" and e="e" and r="r" in allocIn)
  apply assumption
 apply (frule_tac a="e" in searchLessLength)
 apply simp
apply clarsimp
apply (simp add:exec_cons exec_null)
done
```

Till now we have proven that *optCmp* is correct for an expression list that satisfies some properties. Now we show that one such list can be generated from any given expression.

Pre-order traversal of an expression:

```
primrec preOrd :: "exp ⇒ exp list" where
  "preOrd (Const n) = [Const n]"
| "preOrd (Var v) = [Var v]"
| "preOrd (Op e1 e2) = (Op e1 e2)#(preOrd e1 @ preOrd e2)"


lemma self_in_preOrd:"e ∈ set (preOrd e)"
by(case_tac e, auto)
```

The function *optExp* generates a distinct sub-expression list from a given expression:

```
definition optExp ::"exp ⇒ exp list" where
  "optExp e == rev (remdups (preOrd e))"

lemma distinct_rev:"distinct (rev xs) = distinct xs"
by (induct xs, auto)
```

```
lemma supExp_app:"⋀ bs. ⟦ supExp as ; supExp bs ⟧ ⟹ supExp (as @ bs)"
apply (induct as)
 apply clarsimp
apply (case_tac a)
  apply auto
done

lemma supExp_remdups:"⋀ bs. supExp as ⟹ supExp (remdups as)"
apply (induct as)
 apply clarsimp
apply (case_tac a)
  apply auto
done

lemma supExp_preOrd:"supExp (preOrd e)"
apply (induct e)
 apply (auto dest:supExp_app simp:self_in_preOrd )
done
```

Proof that a list generated by *optExp* is distinct and satisfies the sub-expression property:

```
lemma optExpDistinct:"distinct(optExp e)"
by (simp add:optExp_def)

lemma optExpSupExp:"subExp (optExp e)"
apply (induct e)
 apply (auto simp:optExp_def self_in_preOrd subExp_def
        intro:supExp_remdups supExp_preOrd supExp_app)
done
```

Do *optCmp optExp* and generate code that evaluate all common sub-expressions only once?

Yes. Since *optExp* returns all commonly occurring sub-expressions only once, and *optCmp* evaluates these only once, all common sub-expressions are evaluated only once.

But, for those of little faith:

```
lemma opt:"⋀m r. length (filter (λe. ∃x y. e = (Op x y)) es) =
                length (filter (λi. ∃x. i = (OPER x)) (optCmp es m r))"
apply (induct es)
 apply clarsimp
apply (case_tac a)
  apply auto
done
```

end