Isabelle/HOL Exercises
Projects

Compilation with Side Effects

This exercise deals with the compiler example in Section 3.3 of the Isabelle/HOL tutorial. The simple side effect free expressions are extended with side effects.

1. Read Sections 3.3 and 8.2 of the Isabelle/HOL tutorial. Study the section about *fun_upd* in theory *Fun* of HOL: *fun_upd f x y*, written *f(x:=y)*, is *f* updated at *x* with new value *y*.

2. Extend data type *('a, 'v) expr* with a new alternative *Assign x e* that shall represent an assignment *x = e* of the value of the expression *e* to the variable *x*. The value of an assignment shall be the value of *e*.

3. Modify the evaluation function *value* such that it can deal with assignments. Note that since the evaluation of an expression may now change the environment, it no longer suffices to return only the value from the evaluation of an expression.

   Define a function *se_free :: expr ⇒ bool* that identifies side effect free expressions. Show that *se_free e* implies that evaluation of *e* does not change the environment.

4. Extend data type *('a, 'v) instr* with a new instruction *Store x* that stores the topmost element on the stack in address/variable *x*, without removing it from the stack. Update the machine semantics *exec* accordingly. You will face the same problem as in the extension of *value*.

5. Modify the compiler *comp* and its correctness proof to accommodate the above changes.

```
type_synonym 'v binop = "'v ⇒ 'v ⇒ 'v"

datatype ('a, 'v) exp
  = Const 'v
  | Var 'a
  | Binop "'v binop" "('a, 'v) exp" "('a, 'v) exp"
  | Assign 'a "('a, 'v) exp"

primrec val :: "('a, 'v) exp => ('a => 'v) => 'v * ('a => 'v)" where
```

```
  "val (Const c)       env = (c, env)"
| "val (Var x)         env = (env x, env)"
| "val (Binop f e1 e2) env =
     (let (x, env1) = val e1 env;
          (y, env2) = val e2 env1
      in (f x y, env2))"
| "val (Assign a e)    env =
     (let (x, env') = val e env
      in (x, env' (a := x)))"
```

```
primrec se_free :: "('a, 'v) exp ⇒ bool" where
  "se_free (Const c)       = True"
| "se_free (Var x)         = True"
| "se_free (Binop f e1 e2) = (se_free e1 ∧ se_free e2)"
| "se_free (Assign x e)    = False"
```

```
lemma "se_free e ⟶ snd (val e env) = env"
  apply (induct_tac e)
     apply simp
    apply simp
   apply (simp add: Let_def split_def)
  apply simp
  done
```

```
datatype ('a, 'v) instr
  = CLoad 'v
  | VLoad 'a
  | Store 'a
  | Apply "'v binop"
```

```
primrec exec :: "('a, 'v) instr list ⇒ ('a ⇒ 'v) ⇒ 'v list ⇒ 'v list × ('a
⇒ 'v)" where
  "exec [] hp vs = (vs, hp)"
| "exec (i#is) hp vs = (case i of
    CLoad v ⇒ exec is hp (v#vs)
  | VLoad a ⇒ exec is hp (hp a#vs)
  | Store a ⇒ exec is (hp (a:= hd vs)) vs
  | Apply f ⇒ exec is hp (f (hd (tl vs)) (hd vs)#(tl (tl vs))))"
```

```
lemma
  "exec [CLoad (3::nat),
         VLoad x,
         CLoad 4,
```

```
        Apply (op *),
        Apply (op +)]
       (λx. 0) [] = ([3], λx. 0)"
  by simp

primrec comp :: "('a, 'v) exp ⇒ ('a, 'v) instr list" where
  "comp (Const c) = [CLoad c]"
| "comp (Var x) = [VLoad x]"
| "comp (Assign x e) = (comp e) @ [Store x]"
| "comp (Binop f e1 e2) = (comp e1) @ (comp e2) @ [Apply f]"

lemma [simp]:
  "∀hp vs. exec (xs@ys) hp vs = (let (vs', hp') = exec xs hp vs in exec ys hp'
vs')"
  apply (induct_tac xs)
  apply (simp add: Let_def)
  apply (simp add: Let_def split: instr.split)
  done

theorem [simp]:
  "∀vs hp. exec (comp e) hp vs = ([fst (val e hp)] @ vs, snd (val e hp))"
  apply (induct_tac e)
     apply simp
    apply simp
   apply (simp add: Let_def split_def)
  apply (simp add: Let_def split_def)
  apply (simp add: fun_upd_def)
  done

corollary "exec (comp e) s [] = ([fst (val e s)], snd (val e s))"
  by simp
```