Fundamentals of MCS: CS - Part 2

NGUYEN T. Hoang - SID: 15M54097

Fall 2015, W832 Mon. Period 7-8
Due date: 2016/01/18

Problem

For this Part II of Fall 2015 Fundamentals of Mathematical and Computing Sciences: Computer Science class, I choose **Assignment 3** for submission.

Q3.1. We would like to port the compiler to another stack machine whose behavior is slightly different from the original one. Although the representation of its structure remains the same (`Definition prog := list instr` and `Definition stack := list nat`), the new stack machine's interpretation of instructions is slightly different:

```
Definition instrDenote' (i : instr) (s : stack) : option stack :=
  match i with
  | iConst n ⇒ Some (n :: s)
  | iBinop b ⇒ match s with
               | arg2 :: arg1 :: s' ⇒ Some ((binopDenote b) arg1 arg2 :: s')
               | _ ⇒ None
               end
  end.
```

The instrDenote' function assumes that the second operand at the stack top while instrDenote assumes the first one at the top. Given this modified instrDenote' function, try to modifiy the implementation of the compiler so that it suits the new definition and prove its correctness.

Q3.2. Extend your implementation of **Q3.1** to add Minus operator to binop and adjust definitions of denotations, the compiler, appropriately and complete the proof.

Answer:

Q3.1 - Modified Stack Machine.

Since we are given new instrDenote' function, I am going to change the compile and progDenote function into compile' and progDenote' function that accept the new definition of instrDenote'. The new functions are defined as follow:

```
Fixpoint progDenote' (p : prog) (s : stack) : option stack :=
  match p with
  | nil ⇒ Some s
  | i :: p' ⇒ match instrDenote' i s with
                 | None ⇒ None
                 | Some s' ⇒ progDenote' p' s'
                 end
  end.
```

```
Fixpoint compile' (e : exp) : prog :=
  match e with
  | Const n ⇒ iConst n :: nil
  | Binop b e1 e2 ⇒ (compile' e1) ++ (compile' e2) ++ (iBinop b :: nil)
  end.
```

Before going to the proof, I would like to test out the new Stack Machine with few examples of program evaluation and compiler evaluation:

```
Eval simpl in progDenote' (compile' (Const 3)) nil.
   = Some (3 :: nil) : option stack
```

```
Eval simpl in progDenote' (compile' (Binop Plus (Const 3) (Const 4))) nil.
   = Some (7 :: nil) : option stack
```

```
Eval simpl in progDenote' (compile' (Binop Times
                  (Binop Plus (Const 3) (Const 4))
                  (Binop Plus (Const 5) (Const 6)))) nil.
   = Some (77 :: nil) : option stack
```

```
Eval simpl in compile' (Binop Times (Binop Plus (Const 2) (Const 3)) (Const 7)).
   = iConst 3 :: iConst 2 :: iBinopPlus :: iConst 7 :: iBinop Times :: nil : prog
```

Our modified compiler should work with *all* input, therefore we have the compiple'_correct theorem as follow:

```
Theorem compile'_correct : ∀ e, progDenote' (compile' e) nil = Some (expDenote e :: nil).
```

To prove this theorem, as in {cpdt}, I will use the standard trick of *strengthening the induction hypothesis*. By proving the fact that, given *any* expression, program list state, and stack state, the modified compiler will correctly compile the program to run with progDenote'.

```
Lemma compile'_correct' : ∀ e p s,
  progDenote' (compile' e ++ p) s = progDenote' p (expDenote e :: s).
```

```
1 subgoal

  ==============================
   ∀ (e : exp) (p : list instr) (s : stack),
   progDenote' (compile' e ++ p) s = progDenote' p (expDenote e :: s)
```

A typical strategy for handling "∀" is to use `intros` tactic. However, if we use `intros` now, before performing `induction` on expression e, we will have some problem with Coq cannot recognize some pattern later. Therefore, the tactic `induction` will be used to break down expression e into basic cases first, then I will apply `intros` tactic for each case.

```
induction e.
```

```
2 subgoals
  n : nat
  ==============================
   ∀ (p : list instr) (s : stack),
   progDenote' (compile' (Const n) ++ p) s =
   progDenote' p (expDenote (Const n) :: s)
subgoal 2 is:
 ∀ (p : list instr) (s : stack),
 progDenote' (compile' (Binop b e1 e2) ++ p) s =
 progDenote' p (expDenote (Binop b e1 e2) :: s)
```

Assuming we are given some arbitary stack and program:

```
intros.
```

```
2 subgoals
  n : nat
  p : list instr
```

s : stack

==============================

progDenote' (compile' (Const n) ++ p) s =
progDenote' p (expDenote (Const n) :: s)

subgoal 2 *is*:

∀ (p : **list instr**) (s : stack),
progDenote' (compile' (Binop b $e1$ $e2$) ++ p) s =
progDenote' p (expDenote (Binop b $e1$ $e2$) :: s)

The first subgoal can be proved by simplify the function compile' and expDenote. The tactic named `simpl` and `reflexivity` does exactly what we want. `simpl`.

2 subgoals

n : **nat**
p : **list instr**
s : stack

==============================

progDenote' p (n :: s) = progDenote' p (n :: s)

subgoal 2 *is*

∀ (p : **list instr**) (s : stack),
progDenote' (compile' (Binop b $e1$ $e2$) ++ p) s =
progDenote' p (expDenote (Binop b $e1$ $e2$) :: s)

By using simple `reflexivity` tactic, I have proved the first subgoal.

`reflexivity`.

1 subgoal

b : **binop**
$e1$: **exp**
$e2$: **exp**
$IHe1$: progDenote' (compile' $e1$ ++ p) s = progDenote' p (expDenote $e1$ :: s)
$IHe2$: progDenote' (compile' $e2$ ++ p) s = progDenote' p (expDenote $e2$ :: s)

==============================

∀ (p : **list instr**) (s : stack),

progDenote' (compile' (Binop b $e1$ $e2$) ++ p) s =
progDenote' p (expDenote (Binop b $e1$ $e2$) :: s)

Here we have *IHe1* and *IHe2* as two inductive hypothesis. By making the same assumption to handle with "∀", we have:

```
intros.
```

```
1 subgoal
```

b : **binop**
$e1$: **exp**
$e2$: **exp**
IHe1 : progDenote' (compile' $e1$ ++ p) s = progDenote' p (expDenote $e1$ :: s)
IHe2 : progDenote' (compile' $e2$ ++ p) s = progDenote' p (expDenote $e2$ :: s)
p : **list instr**
s : stack
============================
progDenote' (compile' (Binop b $e1$ $e2$) ++ p) s =
progDenote' p (expDenote (Binop b $e1$ $e2$) :: s)

The tactic `simpl` will evaluate the compile' and expDenote functions:

```
simpl.
```

```
1 subgoal
```

b : **binop**
$e1$: **exp**
$e2$: **exp**
IHe1 : progDenote' (compile' $e1$ ++ p) s = progDenote' p (expDenote $e1$ :: s)
IHe2 : progDenote' (compile' $e2$ ++ p) s = progDenote' p (expDenote $e2$ :: s)
p : **list instr**
s : stack
============================
progDenote' ((compile' $e1$ ++ compile' $e2$ ++ iBinop b :: nil) ++ p) s =
progDenote' p (binopDenote b (expDenote $e1$) (expDenote $e2$) :: s)

To make the LHS of our target goal similar to the first inductive hypothesis *IHe1*, I will apply the reverse association rule for **list** concatenation.

```
Check app_assoc_reverse.
```

app_assoc_reverse
  : ∀ (A : Type) (l m n : **list** A), (l ++ m) ++ n = l ++ m ++ n

```
rewrite app_assoc_reverse.
```

1 subgoal

b : **binop**
$e1$: **exp**
$e2$: **exp**
$IHe1$: progDenote' (compile' $e1$ ++ p) s = progDenote' p (expDenote $e1$ :: s)
$IHe2$: progDenote' (compile' $e2$ ++ p) s = progDenote' p (expDenote $e2$ :: s)
p : **list instr**
s : stack
==============================
progDenote' (compile' $e1$ ++ (compile' $e2$ ++ iBinop b :: nil) ++ p) s =
progDenote' p (binopDenote b (expDenote $e1$) (expDenote $e2$) :: s)

Now we can apply the inductive hypotheses to "push" $e1$ and $e2$ of the LHS to the LHS stack.

```
rewrite IHe1.
```

1 subgoal

b : **binop**
$e1$: **exp**
$e2$: **exp**
$IHe1$: progDenote' (compile' $e1$ ++ p) s = progDenote' p (expDenote $e1$ :: s)
$IHe2$: progDenote' (compile' $e2$ ++ p) s = progDenote' p (expDenote $e2$ :: s)
p : **list instr**
s : stack
==============================
progDenote' ((compile' $e2$ ++ iBinop b :: nil) ++ p) (expDenote $e1$ :: s) =
progDenote' p (binopDenote b (expDenote $e1$) (expDenote $e2$) :: s)

```
rewrite app_assoc_reverse.
```

rewrite *IHe2*.

1 subgoal

b : **binop**
$e1$: **exp**
$e2$: **exp**
$IHe1$: progDenote' (compile' $e1$ ++ p) s = progDenote' p (expDenote $e1$ :: s)
$IHe2$: progDenote' (compile' $e2$ ++ p) s = progDenote' p (expDenote $e2$ :: s)
p : **list instr**
s : stack
============================
progDenote' ((iBinop b :: nil) ++ p) (expDenote $e2$ :: expDenote $e1$ :: s) =
progDenote' p (binopDenote b (expDenote $e1$) (expDenote $e2$) :: s)

At this step, we can use the `simpl` tactic again since it is trivial to evaluate the LHS's progDenote' with iBinop p :: nil.

simpl.

1 subgoal

b : **binop**
$e1$: **exp**
$e2$: **exp**
$IHe1$: progDenote' (compile' $e1$ ++ p) s = progDenote' p (expDenote $e1$ :: s)
$IHe2$: progDenote' (compile' $e2$ ++ p) s = progDenote' p (expDenote $e2$ :: s)
p : **list instr**
s : stack
============================
progDenote' p (binopDenote b (expDenote $e1$) (expDenote $e2$) :: s) =
progDenote' p (binopDenote b (expDenote $e1$) (expDenote $e2$) :: s)

I comple the proof of this lemma by `reflexivity` and save it with `Qed`.

reflexivity.
Qed.

introduction e.

```
intros.
simpl.
reflexivity.

intros.
simpl.
rewrite app_assoc_reverse.
rewrite IHe1.
rewrite app_assoc_reverse.
rewrite IHe2.
simpl.
reflexivity.
```

compile'_correct' *is defined*

Now we can go back to prove the main theorem:

```
Theorem compile'_correct : ∀ e, progDenote' (compile' e) nil = Some (expDenote e :: nil).
```

Just like with the lemma compile'_correct', I will firstly introduce the expression e and then append nil to e so that the LHS has the form of compile'_correct'.

```
intros.
rewrite (app_nil_end (compile' e)).
```

```
1 subgoal
  e : exp
  ==============================
   progDenote' (compile' e ++ nil) nil = Some (expDenote e :: nil)
```

The theorem is proved by applying lemma compile'_correct' and *reflexivility*.

```
rewrite compile'_correct'.
```

```
1 subgoal
  e : exp
  ==============================
   progDenote' nil (expDenote e :: nil) = Some (expDenote e :: nil)
```

```
reflexivity.
```

```
Qed.

 intros.
 rewrite (app_nil_end (compile' e)).
 rewrite compile'_correct'.
 reflexivity.
compile'_correct is defined
```

Q3.2 - Extended Stack Machine.

The new Stack Machine is defined in module ext as follow: (I keep the definition of stack since it is not necessary to re-define it.

```
Module EXT.

Require Import List.

Inductive binop : Set := Plus | Times | Minus.
Definition binopDenote (b:binop) : nat → nat → nat :=
  match b with
  | Plus ⇒ plus
  | Times ⇒ mult
  | Minus ⇒ minus
  end.
Inductive exp : Set :=
  | Const : nat → exp
  | Binop : binop → exp → exp → exp.
Fixpoint expDenote (e:exp) : nat :=
  match e with
  | Const n ⇒ n
  | Binop b e1 e2 ⇒ (binopDenote b) (expDenote e1) (expDenote e2)
  end.
Inductive instr : Set :=
  | iConst : nat → instr
  | iBinop : binop → instr.
Definition prog := list instr.
Definition instrDenote (i : instr) (s : stack) : option stack :=
  match i with
  | iConst n ⇒ Some (n :: s)
  | iBinop b ⇒ match s with
```

```
              | arg2 :: arg1 :: s' ⇒ Some ((binopDenote b) arg1 arg2 :: s')
              | _ ⇒ None
              end
  end.
Fixpoint progDenote (p : prog) (s : stack) : option stack :=
  match p with
  | nil ⇒ Some s
  | i :: p' ⇒ match instrDenote i s with
             | None ⇒ None
             | Some s' ⇒ progDenote p' s'
             end
  end.
Fixpoint compile (e : exp) : prog :=
  match e with
  | Const n ⇒ iConst n :: nil
  | Binop b e1 e2 ⇒ (compile e2) ++ (compile e1) ++ (iBinop b :: nil)
  end.
End EXT.
```

Q1.1 - Regular language definition with $\approx_L$.

My proof contains two parts: