Contents

Chapter 1

A Programming Language

1.1 Defining a program

Our development of computability theory will be based on a specific programming language $\mathscr{S}$. We will use certain letters as variables whose values are natural numbers. There are three kinds of variables separated by the letters Y, X and Z to represent the output, input variables and local variables.

```
Inductive var: Type:=
  | Y | X(i:nat) | Z(i:nat).
```

In addition to variables, there are labels that are used to commute between different instructions of a program. These labels are exhibited by certain letters and an index[1]. The purpose of this separation is only to facilitate programming in practice.

```
Inductive lab: Type:=
  | A(i:nat) | B(i:nat) | C(i:nat) | D(i:nat) | E(i:nat).
```

Combining variables and labels, there are four different kinds of statements:

- incrementing the value of a specified variable
- decrementing the value of a specified variable (if it's 0 no changes happen)
- simply do nothing to a specified variable
- if the value of a specified variable is not 0 then go to the first statement which is labeled as specified (this statement will be clarified after defining programs)

```
Inductive statement: Type:=
  | inc(v:var) | dec(v:var) | skip(v:var) | cond(v:var)(l: lab).
```

[1]In the original reference [1], all indices for variables and labels start from 1 not 0. But since *nat* is a more canonical choice for our formalization, we preferred to start from 0.

We label statements and call them instructions. It's also possible to leave some statements unlabeled but use them as instructions for a program.

```
Inductive ins: Type:=
  | l_s (l: lab)(s: statement)
  | __s (s: statement).
```

A program is simply a list of instructions.

```
Definition prog: Type:= list ins.
```

In the next chapter, we see how a program is assumed to work in some examples. Then we formally define the computation of a program in Chapter 2.

1.2 Examples of programs

Before beginning to write programs, we introduce the following notations:

- v ++ is used for *inc v*
- v – is used for *dec v*
- $v \to l$ is used for *cond v l*

The following program (more strictly, program scheme) collapses the value of some arbitrary variable v to 0.

```
Definition var_clear (l:lab)(v:var):prog :=
  <{
    [ l ] v -;
    [   ] v → l
  }>.
Notation "<[ l ] v ← 0 >":= (var_clear l v) (in custom SProg at level 100).
```

Another useful program could do the following: "increasing the value of some variable to the addition of its current value and the value of some other program". Here is an implementation:

```
Definition var_transmit (l1 l2 e :lab)(z v v':var):prog :=
  <{ # If v' has the value 0 then nothing has to be done.
     [    ] v' → l1  ;
     [    ] z ++ ;
     [    ] z → e;
     # otherwise, the first step is to empty v' in both z and v
     [ l1 ] v' - ;
     [    ] v ++ ;
     [    ] z ++;
     [    ] v' → l1;
     # next, retrieving the value of v' from z
     [ l2 ] z - ;
     [    ] v' ++ ;
     [    ] z → l2;
     [ e ] z -
  }>.
```

Note that the program works correctly if the labels and variables are all distinct.

Now we might think of instantiating *var_transmit* on Y and X 0 and then concatenate it to another instantiation on Y and X 1 to create a program that computes the addition function. This is a good idea, but we have to be careful about the names we choose for the other parameters, i.e. the labels and the variable z. A good choice is as follows:

```
Definition addition1:prog := <{
  var_transmit (A 0) (B 0) (E 0) (Z 0) Y (X 0);
  var_transmit (A 1) (B 1) (E 1) (Z 1) Y (X 1)
}>.
```

We shall mention that the value of the output and the local variables is assumed to be 0 at the beginning. Now observing what this program does in the first part, we notice that the value of Z 0 is always 0 after termination. Thus we can reuse it for the second part and eliminate Z 1:

```
Definition addition2:prog := <{
  var_transmit (A 0) (B 0) (E 0) (Z 0) Y (X 0);
  var_transmit (A 1) (B 1) (E 1) (Z 0) Y (X 1)
}>.
```

Although the real extended version of the program can be simplified even more, we won't go further and continue with the current version. It's also worth mentioning that as we combined programs carefully to prevent conflict, it's possible to use any program in another one in a more general form. But it's clear that in complex examples it can be too hard to manually handle the conflictions. Therefore, we formally define a machinery to automate this process in Chapter 3.

1.3 Coding of variable and labels

To be able to easily manipulate variables and labels and to prove facts about them, we encode them in natural numbers and decode them whenever needed. Here is the coding and decoding of variables:

```
Definition code_var(v: var):=
  match v with
  | Y ⇒ 0
  | X i ⇒ 2×i+1
  | Z i ⇒ 2×i+2
  end.
```

```
Fixpoint decode_var(n:nat):=
  match n with
  | 0 ⇒ Y
  | S i ⇒ next_var [2] (decode_var i)
  end.
```

And here is the coding and decoding of labels [3] :

```
Definition code_lab(l: lab):=
  match l with
  | A i ⇒ i×5
  | B i ⇒ i×5 + 1
  | C i ⇒ i×5 + 2
  | D i ⇒ i×5 + 3
  | E i ⇒ i×5 + 4
  end.
```

```
Fixpoint decode_lab(n:nat):=
  match n with
  | 0 ⇒ A 0
  | S i ⇒ next_lab [4] (decode_lab i)
  end.
```

From now on, we use $\# \ x$ to show the coding of x, regardless of x is a variable or a label.

[2]simply the next variable with regard to it's code.

[3]In the original reference [1] , coding of labels start from 1 not 0. But since *nat* is a more canonical choice for our formalization, we preferred to start from 0.

[4]simply the next label with regard to it's code.

Chapter 2

Computation of a Program

2.1 Updating and storing information

To use programs for computation, we have to store the values of variables after executing each instruction. Thus we use a list of numbers called *state*.

```
Definition state:= list nat.
```

The *state_var* function simply shows the value of a variable in a state based on its code and regards the last value's index as 0 (thus representing the output) and the head represents the value of the greatest variable regarding its code. Also, we assume that the default value of all variables is 0, especially those that are not represented in the state.

```
Example state_var_test1 : state_var [1;2;3;4;5] Y = 5 .
Proof. reflexivity. Qed.
Example state_var_test2 : state_var [1;2;3;4;5] (Z 1) = 1 .
Proof. reflexivity. Qed.
Example state_var_test3 : state_var [1;2;3;4;5] (X 2) = 0 .
Proof. reflexivity. Qed.
Example state_var_test4 : state_var nil Y = 0 .
Proof. reflexivity. Qed.
```

Next we have the *var_update_state* function which updates the value of a variable in a state as desired. As we mentioned before, the default value of all variables is 0, thus if this variable is not seen in the state yet, first the value of all hidden variables (0) between this variable and the head of the state should emerge. The following examples clarify this matter:

```
Example var_update_state_test1: var_update_state Y 5 nil = [5].
Proof. reflexivity. Qed.

Example var_update_state_test2 : var_update_state (X 2) 3 nil = [3;0;0;0;0;0].
Proof. reflexivity. Qed.
```

```
Example var_update_state_test3 :
var_update_state (Z 1) 4 (var_update_state (X 2) 3 nil) = [3;4;0;0;0;0].
Proof. reflexivity. Qed.
```

```
Example var_update_state_test4 :
var_update_state (X 2) 4 (var_update_state (X 2) 3 nil) = [4;0;0;0;0;0].
Proof. reflexivity. Qed.
```

The theorem below states that the implemented algorithm for updating the value of a variable in a state, actually does this task.

```
Theorem var_update_state_updates: ∀ v: var, ∀ s: state, ∀ n: nat,
  state_var (var_update_state v n s) v = n.
```

The proof of the theorem uses some lemmas about the details of the implementation, which are rather tedious, thus we won't mention them here.

2.2 The evaluation procedure

Based on the possible instructions in a program, we can update a state only in the two following manner:

```
Definition var_increment_state (v: var) (s: state):=
  var_update_state v (S (state_var s v )) s.
```

```
Definition var_decrement_state (v: var) (s: state):=
  var_update_state v ((state_var s v ) - 1) s.
```

Soundness of these two functions immediately follows from *var_update_state_updates*:

```
Theorem var_increment_state_increments: ∀ v: var, ∀ s: state,
  state_var (var_increment_state v s) v = S (state_var s v).
Proof. intros. unfold var_increment_state. rewrite var_update_state_updates. lia.
Qed.
```

```
Theorem var_decrement_state_decrements: ∀ v: var, ∀ s: state,
  state_var (var_decrement_state v s) v = (state_var s v) -1.
Proof. intros. unfold var_decrement_state. rewrite var_update_state_updates. lia.
Qed.
```

Now we shall define the effect of a statement on a state:

```
Definition statement_on_state (st: statement) (s: state):=
  match st with
  | inc v ⇒ var_increment_state v s
  | dec v ⇒ var_decrement_state v s
  | _ ⇒ s
  end.
```

Here is an example:

```
Example statement_on_state_test1 :
  statement_on_state (<{(Z 0) ++}>) nil = [1;0;0].
Proof. reflexivity. Qed.
```

The next step is to formalize the notion of commuting between the instructions (lines) of a program. Thus in addition to the state, we store the number of the line of the current instruction and call this pair an *snapshot*:

```
Inductive snapshot:= | line_state (i: nat) (s: state).
Definition line_ (ls: snapshot):= match ls with | line_state i s ⇒ i end.
Definition _state (ls: snapshot):= match ls with | line_state i s ⇒ s end.
Notation " < l >- s ":= (line_state l s) (at level 200).
```

Regarding a program, we call any snapshot with a line number equal to *length* $p+1$ a terminal snapshot.

```
Definition terminal_snap (p: prog) (sn: snapshot):= line_ sn =? length p + 1.
```

The mechanism of computing the *successor* of a *non-terminal snapshot* namely $<i>-s$ by a program is as follows:

- case 1) If the ith instruction is increment, decrement or skip a variable, then do the corresponding statement on s and increase i by 1.
- case 2) If the ith instruction is a conditional for the variable v and label l, then
  - 2a) If the value of v is 0 in s, then only increase i by 1.
  - 2b) If the value of v is not 0 in s, then find the first line in the program labeled l and replace i with its line number. If there is no such label, replace i with *length* $p+1$.

The successor of a *terminal snapshot* is assumed to be itself.

```
Definition successor_sn (sn: snapshot) (p: prog):=
  if (length p) <? line_ sn
    then sn
    else line_state
          // line after one iteration
              (match (_s (ith_ins_prog (line_ sn) p)) with
              | cond v l ⇒ match state_var (_state sn) v with
                | 0 ⇒ (line_ sn) + 1
                | _ ⇒ line_finder l p
                end
              | _ ⇒ (line_ sn) + 1
              end)
          // state after one iteration
              (statement_on_state
                (_s (ith_ins_prog (line_ sn) p))
                (_state sn)).
```

The following function repeats this procedure for arbitrarily n number of times:

```
Fixpoint n_iteration (sn: snapshot) (p: prog) (n: nat):=
  match n with
  | 0 ⇒ sn
  | S k ⇒ successor_sn (n_iteration sn p k) p
  end.
Notation "s →<{ p | × n } ":= (n_iteration s p n) (at level 50, left associativity).
Notation "s →<{ p | × n }> " :=
  (ith_val 0 (_state (n_iteration s p n))) (at level 50).
```

Recall the *var_clear* program: l v −; v → l The following examples show how an instantiation of this program reduces the value of some variable and then terminates, step by step:

```
Example successor_sn_test0:
  (<1>-[2;0]) →<{ <{<[A 0] X 0 ← 0>}> | × 0 }= <1>-[2;0].
Proof. reflexivity. Qed.
Example successor_sn_test1:
  (<1>-[2;0]) →<{ <{<[A 0] X 0 ← 0>}> | × 1 }= <2>-[1;0].
Proof. reflexivity. Qed.
Example successor_sn_test2:
  (<2>-[1;0]) →<{ <{<[A 0] X 0 ← 0>}> | × 1 }= <1>-[1;0].
Proof. reflexivity. Qed.
Example successor_sn_test3:
  (<1>-[1;0]) →<{ <{<[A 0] X 0 ← 0>}> | × 1 }= <2>-[0;0].
Proof. reflexivity. Qed.
Example successor_sn_test4:
    (<2>-[0;0]) ⇢<{ <{<[A 0] X 0 ← 0>}> | × 1 }= <3>-[0;0].
  Proof. reflexivity. Qed.
  Example successor_sn_test5:
    (<3>-[0;0]) ⇢<{<{ <[A 0] X 0 ← 0>}> | × 1 }= <3>-[0;0].
  Proof. reflexivity. Qed.
```

Or simply we can compute:

```
  Example n_iteration_test1 :
    (<1>-[2;0]) ⇢<{ <{<[A 0] X 0 ← 0>}> | × 5 }>= 0.
  Proof. reflexivity. Qed.
```

Another example is to test the *addition2* on some examples:

```
  Example length_of_addition2: (length addition2) = 22.
  Proof. reflexivity. Qed.

  Example n_iteration_addition2_test1 :
    (<1>-[5;0;3;0]) ⇢<{ addition2 | × 60 }= <23>-[5;0;3;8].
  Proof. reflexivity. Qed.
```

The value of all variables is as desired, and the line number is correctly 23 which shows the program has been terminated.

2.3 Computation and related notions

A computation of a program is a list of snapshots, each of them (except the first one) being the successor of the previous one and the last one being a terminal snapshot:

```
Definition head_snap (snl: list snapshot):=
  match snl with
  | nil ⇒ <0>-nil  (can't be the successor of any snapshot)
  | cons sn l ⇒ sn
  end.

Inductive computation : prog → list snapshot → Prop :=
  | emp_comp (p: prog): computation p nil
  | final_comp (p: prog) (sn: snapshot):
     (terminal_snap p sn = true) → computation p [sn]
  | body_comp (p: prog) (sn: snapshot ) (sl: list snapshot):
     computation p sl → sn ⇢<{p |× 1}= (head_snap sl)
     → computation p (sn::sl).
```

The *halt* predicate is expressed as follows:

```
Definition halt (p: prog) (s: snapshot):= ∃ n: nat,
  terminal_snap p (n_iteration s p n) = true.
```

We call two programs *a_equal* if their effect is the same on any snapshot.

```
Definition a_equal (p p':prog) := ∀ s:snapshot,
  successor_sn s p = successor_sn s p'.
```

For example, any program is *a_equal* to the result of renaming of its labels (but not vice versa). However this doesn't hold for the renaming of variables. We'll use these renamings in the next chapter while formalizing macros.

Chapter 3

Combining programs

Simple instructions of $\mathscr{S}$ programs, make it easy to reason about them, but on the other hand, it's too laborious to write complex programs in practice. To tackle this problem we introduce "macro" instructions, to use previously defined programs in constructing the new ones. As we saw previously, it's not as simple as just inserting a predefined program somewhere in the body of the new one. Clearly, doing so arises conflicts between their labels and variables and the result won't be as desired. Thus first we must process the program such that without any conflict, it does the same task as it did before. Then add extra instructions to the main program to make it able to run the modified program on some inputs and extract the output. Finally, We abbreviate the calls to this program as a special command and call it a macro. There are two main steps in preprocessing a program to make it ready for insertion in any other program as described in the textbook. In the next two sections, we discuss and formally implement any of these procedures.

3.1 Normalizing a program

Changing the name of the labels of a program so that they will be among $A\ 0$, $A\ 1$,..., $A\ l$ except the labels in the instruction which do not appear as the labels of any line (thus basically an exit label); these labels are all replaced with E 0. Clearly, renaming of labels won't change the task of the program. In fact, the effect of the program on any snapshot remains the same. But the reason for doing these will be clarified as we go through the next steps.

The following function takes a program and checks the last *index* number of the lines and each time if the line was labeled of any form other than $A\ i$, substitutes its label in the whole program with the first label of the form $A\ i$ which doesn't exist in the program yet (*max_A_lab* finds such label and *subst_lab* replaces with it). Thus it's enough to start this function with $index = length\ p$ to change all line labels to the desired to form.

```
Fixpoint general_all_line_lab_A (p: prog) (index: nat):=
  match index with
    | 0 ⇒ p
    | S i ⇒ match ith_ins_prog index p with
      | __s s ⇒ general_all_line_lab_A p i
      | l_s l s ⇒ match l with
        | A j ⇒ general_all_line_lab_A p i
        | _ ⇒ general_all_line_lab_A (subst_lab p l (max_A_lab p)) i
        end
      end
  end.
Definition all_line_lab_A (p: prog):= general_all_line_lab_A p (length p).
```

Next we need to normalize the exit labels. The following functions work similarly to the previous ones, except they revise the exit labels.

```
Fixpoint general_E_only_exit (p: prog) (index: nat):=
  match index with
    | 0 ⇒ p
    | S i ⇒ match ith_ins_prog i p with
      | __s (cond v l) ⇒ if is_in_lab_list l (lines_lab p)
                           then general_E_only_exit p i
                           else general_E_only_exit (subst_lab p l (E 0)) i
      | l_s _ (cond v l) ⇒ if is_in_lab_list l (lines_lab p)
                           then general_E_only_exit p i
                           else general_E_only_exit (subst_lab p l (E 0)) i
      | _ ⇒ general_E_only_exit p i
      end
  end.
Definition E_only_exit (p:prog) := general_E_only_exit p (length p + 1).
```

Finally, we encapsulate both of the aforementioned functions into one:

```
Definition normalize (p: prog):= E_only_exit (all_line_lab_A p).
Definition both_valued := <{
  [ A 2 ] X 0 → B 1;
  [     ] Z 0 ++;
  [     ] Z 0 → C 1;
  [ B 1 ] X 1 → E 0;
  [     ] Z 1 ++;
  [     ] Z 1 → D 2;
  [ E 0 ] Y ++
}>.
Example normalize_test : normalize both_valued = <{
  [ A 2 ] X 0 → A 4;
  [     ] Z 0 ++;
  [     ] Z 0 → E 0;
  [ A 4 ] X 1 → A 3;
  [     ] Z 1 ++;
  [     ] Z 1 → E 0;
  [ A 3 ] Y ++
}>.
Proof. reflexivity. Qed.
```

3.2 Localizing a program

The next step to preparing a program to be inserted in another one is localization. We need to change its variables so that all of them act as local variables. Furthermore, we save the new names of the original input and output variables somewhere, so that later on we'll able to successfully execute the program in another program. It's also necessary to make changes so that after finishing the task, it won't exist the whole main program. Thus (assuming it's a normal program) we add a superfluous instruction labeled $E\ 0$ to play the role of the finish line (and the only way to exit).

For simplicity, we use states for saving the names (more precisely, the coding) of the replaced local variables and pair such state with the corresponding program:

```
Inductive local_prog_inout:= | lp_inout (lp: prog) (input: state).
```

What follows is a general function which in addition to a program, takes a local variable (*zfloor*), an *index* and a *state*. The task is to start from the last line of the program and replace the non-local variable with the given local variable (in the whole program, using *subst_var*) and save the code of the new names of the variables in the given state, and repeat this procedure for *index* times. Thus if we append the aforementioned superfluous instruction and let *zfloor* be the first non-occured local variable of the program (*max_Z_var p*), *index*:= *length p* and *st*:= *nil* then The result would be the desired localized program.

```
  Fixpoint general_localize (p: prog) (zfloor: var) (index: nat) (inout: state):=
    match index with
    | 0 ⇒ lp_inout p inout
    | S k ⇒ match v_ (_s (ith_ins_prog index p)) with
      | Z j ⇒ general_localize p zfloor k inout
      | v ⇒ general_localize
        (subst_var p v zfloor)
        (next_Z zfloor)
        k
        (var_update_state v (# zfloor) inout)
      end
    end.
Definition localize (p: prog):=
  general_localize (<{p ; [ E 0 ] skip Y}>) (max_Z_var p) (length p) nil.
```

As mentioned at the beginning of this section, in order to localization make sense, the program to be localized should be normal at first (o.w. the superfluous instruction $<\{[\ E\ 0\ ]\ skip\ \ Y\}>$ won't act as the only finish line). Thus combining localization with normalization we define the following:

```
Definition local_normal (p:prog):= localize( normalize p).
Example local_normal_test1 : local_normal both_valued = lp_inout <{
   [ A 2 ] Z 4 → A 4;
   [     ] Z 0 ++;
   [     ] Z 0 → E 0;
   [ A 4 ] Z 3 → A 3;
   [     ] Z 1 ++;
   [     ] Z 1 → E 0;
   [ A 3 ] Z 2 ++;
   [ E 0 ] skip (Z 2)
}> [# Z 3; 0; # Z 4; # Z 2 ].
Proof. reflexivity. Qed.
```

3.3 Making programs compatible

Now that we have implemented the machinery for local-normalizing programs it's time to shift the labels (using *lab_shift*) and variables (using *var_shift*) so that no conflict arises when a program is inserted in another one. Furthermore, it's important to preserve the form of the variables and labels so that they remain localized. Two functions *next_eq_even* and *next_eq_5k* take care of this:

```
Definition local_normal_prog (p: prog):=
  match local_normal p with | lp_inout p i ⇒ p end.
```

```
Definition local_normal_inout (p: prog):=
  match local_normal p with | lp_inout p inout ⇒ inout end.

Definition next_eq_even (n: nat):= if even n then n else S n.

Fixpoint next_eq_5k (n: nat):= match n with
  | 0 ⇒ 0 | S ( S (S ( S (S m)))) ⇒ (next_eq_5k m) + 5 | _ ⇒ 5 end.

Definition make_compatible (p: prog) (p': prog):=
  lab_shift
    (var_shift (local_normal_prog p') (next_eq_even(# (max_var p))))
    (next_eq_5k (# (max_lab p))).
```

The following example shows how conflicts are handled:

```
  Example make_compatible1_test1 : make_compatible <{
      [ A 3 ] Z 2 → E 0
    }> both_valued = <{
      [ A 6 ] Z 8 → A 8;
      [     ] Z 4 ++;
      [     ] Z 4 → E 4;
      [ A 8 ] Z 7 → A 7;
      [     ] Z 5 ++;
      [     ] Z 5 → E 4;
      [ A 7 ] Z 6 ++;
      [ E 4 ] skip (Z 6)
    }>.
  Proof. reflexivity. Qed.
```

Similarly the saved input names are needed to be shifted, thus we should do the same on the state. The *state_shift_if_updated* checks what variables are localized and shifts them as much as they fit the compatible program.

```
Definition mac_inout (p: prog) (p': prog):=
  state_shift_if_updated (local_normal_inout p') (next_eq_even(# (max_var p))).

  Example mac_inout_test1 : mac_inout <{
    [ A 3 ] Z 2 → E 0
  }> both_valued = [# Z 7; 0; # Z 8; # Z 6].
  Proof. reflexivity. Qed.
```

3.4 Executing a local program

We formally define a macro to be the tripple of a variable, a program and a list of variables as inputs:

```
Inductive macro:= | vpi (v: var) (p: prog) (i: list var).
```

A macro as an instruction in the middle of a program tells that "run the program p on the inputs i and save its output value in the variable v". To fully unravel what a macro does, despite making the program in macro compatible with the main program, we need to add extra instructions to the main program, so that it can execute the local program. There are three tasks to be done here:

- cleaning the value of all variables of p. This seems reduntand at first glance, but in fact it's necessary if p is going to be executed in a loop
- allocating the values of the variables in the list of inputs i to the input variables of p
- extracting the output of p and save it in w

The function *total_var_clear* does the first job (not so efficiently) while using different labels for each loop of *var_clear*, so no conflicts will arise:

```
Fixpoint total_var_clear (p: prog) (floor: nat):=
  match p with
    | nil ⇒ nil
    | i::p' ⇒ <{
      var_clear (B floor) (ins_var i) ;
      total_var_clear p' (floor + 1)
    }>
  end.
```

The task is done by the *in_allocations* function. It takes a list of variables and the name of local inputs. Then for each variable in the list, based on its rank in the list, the function checks whether the original program takes such input (i.e. whether $X\ rank$ is used in the program or not). If there wasn't such input, it skips this variable, otherwise, it generates instructions to allocate the value of the variable to the corresponding new name of the input and repeats this procedure for all variables in the list. Similar to the previous function, it uses the parameter *floor* to prevent conflicts between labels.

```
Fixpoint in_allocations (lv' : list var)(inout : state)(floor : nat)(w : var):=
  match lv' with
    | nil ⇒ nil
    | v'::l ⇒ if (state_var inout (X (length l))) =? 0
      then (in_allocations l inout floor w)
      else <{
          var_transmit (C floor) (D floor) (E floor)
            w (decode_var (state_var inout (X (length l)))) v';
          \\ recall that w can be reused without making real confliction.
          in_allocations l inout (floor+1) w
        }>
  end.
```

```
Example in_allocations_test1 : in_allocations
[X 2] [# Z 0 ; 0 ] 5 (Z 1) =
<{
  [     ] X 2 → C 5 ;
  [     ] Z 1 ++ ;
  [     ] Z 1 → E 5 ;
  [ C 5 ] X 2 - ;
  [     ] Z 0 ++ ;
  [     ] Z 1 ++ ;
  [     ] X 2 → C 5 ;
  [ D 5 ] Z 1 - ;
  [     ] X 2 ++ ;
  [     ] Z 1 → D 5 ;
  [ E 5 ] Z 1 -
}>.
Proof. reflexivity. Qed.
```

Finally the *unravel_macro* below combines all the instructions and preprocessing and does the third task:

```
Definition lab_index (l: lab) := match l with
  | A i ⇒ i | B i ⇒ i | C i ⇒ i| D i ⇒ i| E i ⇒ i end.

Definition loc_Y (p p': prog):= decode_var(state_var (mac_inout p p') Y).

Definition unravel_macro (p: prog) (m: macro):=
  match m with | vpi w p' inputs ⇒ <{
    <[ B (lab_index (max_lab p) + 2) ] w ← 0 > ;
    total_var_clear (make_compatible p p') (lab_index (max_lab p) + 3);
    in_allocations (rev inputs) (mac_inout p p') (lab_index (max_lab p)+3) w;
    make_compatible p p';
    var_transmit
      (C (lab_index (max_lab p) + 2))
      (D (lab_index (max_lab p) + 2))
      (E (lab_index (max_lab p) + 2))
      (max_Z_var p) w (loc_Y p p')
    }>
  end.
```

3.5 Using macros in practice

Macro lines are of the form $w \leftarrow P(v1,v2,...,vk)$. Currently, we formalized what such a line means in the middle of a program, but we still need to tell coq how to distinguish this lines from the original lines of a real program and then recursively unravel and insert them. To facilitate this process, we stipulate that no unlabeled *skip* v instruction should be used in the main program and the programs in macros (Clearly this is not an actual limitation). This is because the implemented functions to recognize macros, tactically replace them with the commands of the forbidden form while extracting them, so that later on find their place. For example, observe the following function which extracts the lines prior to a presumably separated macro:

```
Fixpoint past_macro (p: prog) (w: var):=
  match p with
  | nil ⇒ nil  (won't happen in practice)
  | __s (skip w) ::p' ⇒ match past_macro p' w with
      | nil ⇒ p'
      | _ ⇒ past_macro p' w
      end
  | _ :: p' ⇒ past_macro p' w
  end.
```

And the following function, uses *past_macro*, once in order and once reversely before and after unraveling the macro, so reconstruct the whole concrete program:

```
Definition ins_mac (p: prog) (m: macro):=
  match m with
  | vpi w _ _ ⇒ (rev (past_macro (rev p) w) ++ (unravel_macro p m) ++ past_macro p w)
  end.
```

Now the function *ins_mac* can be easily generalized to do the same on a list of macros. Furthermore, given a list of instructions containing some macros, it's easy to extract them one by one and then insert their concrete, compatible form to build up a concrete program. We omit the details of such implementation and end this chapter with a real usage of macros in practice:

```
Definition multiplication2:= «<{{
  [ A 0 ] X 0 → A 1,
  [     ] Z 0 ++,
  [     ] Z 0 → E 0,
  [ A 1 ] X 0 -,
  (Z 1) ←<addition2>←( Z 3, X 1),
  var_transmit (B 1) (C 1) (D 1) (Z 2) (Y) (Z 1),
  [     ] Z 0 ++,
  [     ] Z 0 → A 0
}}}>».
```

I discourage you to evaluate the program by asserting examples and using `Proof. reflexivity. Qed.`. It takes *Coq* at least seven minutes to unravel and test it even on small inputs! Instead, use the `Compute` command like as below. After recognizing the program in a few seconds, you can evaluate it on different inputs immediately.

```
Compute (<1>-[5;0;3;0]) →<{ multiplication2 | × 655}>.
```

Bibliography

[1] Martin D. Davis, Ron Sigal, and Elaine J. Weyuker. *Computability, Complexity, and Languages Fundamentals of Theoretical Computer Science.* Morgan Kaufman, San Diego, CA, 2003.