EasyCrypt's *While Language and Hoare Logic*

These slides are an example-based introduction to the features of EasyCrypt's while loop language that correspond to the language we've studied in class so far (and that are used in the notes by Gilles Barthe), as well as to the use of EasyCrypt's Hoare logic.

More information can be found in Sections 2.4–2.5 and 3.4 of the EasyCrypt manual:

https://www.easycrypt.info/documentation/refman.pdf

But note that the manual doesn't have self-contained sections for each of EasyCrypt's logics, and so you'll also find information about EasyCrypt's other program logics in these sections.

The EasyCrypt tactics for Hoare logic are motivated by the ones we've studied in class, but are different in some key ways.

EasyCrypt's *Programming Language*

In EasyCrypt's while language, commands (or statements) are enclosed in *procedures*, which are in turn enclosed in *modules*. Furthermore, modules may have global variables, which their procedures may read and write.

Procedures may call other procedures. But we don't need to make use of this feature at this point in the course. And so consequently we'll ignore for now the Hoare logic tactics for working with procedure calls.

First Example Program

Here is a sample program, which we'll use as our first running example:

```
module M = {
  var x, y : int

  proc f() : unit = {
    if (0 <= x) {
      while (0 < x) {
        x <- x - 1;
        y <- y + 1;
      }
    }
    else {
      while (x < 0) {
        x <- x + 1;
        y <- y - 1;
      }
    }
  }
}.
```

First Example Program

In the above program, the procedure `f` takes in no arguments, and implicitly returns the single element (`()`) of type `unit`. Its assignments are written using `<-`, instead of the `:=` notation used in class. They read and write the global variables `x` and `y` of the module `M`.

We can think of the integers `x` and `y` as the inputs of the program, and of `y` as the program's output. It's not hard to see that the final value of `y` will be equal to the sum of the original values of `x` and `y`.

Hoare Triple for Example Program

Because the variables `x` and `y` are modified during the running of our example program, to state the correctness of the program as a Hoare triple, we need a way of referring to the *original* values of `x` and `y`.

Hoare Triples

Fortunately, we can do this in EasyCrypt using its ambient logic:

```
lemma correct (x_ y_ : int) :
  hoare[M.f : M.x = x_ /\ M.y = y_ ==> M.y = x_ + y_].
proof.
...
qed.
```

The lemma is parameterized by mathematical variables `x_` and `y_`, which are intended to be the initial values of the program's inputs. Its conclusion is EasyCrypt's expression of a Hoare triple. The program is `M.f`. The precondition

```
M.x = x_ /\ M.y = y_
```

assumes that the values of `M.x` and `M.y` are `x_` and `y_`, respectively. And the postcondition

```
M.y = x_ + y_
```

requires that the final value of `M.y` be the sum of `x_` and `y_`.

Proof of First Example

When we begin proving our lemma, we have the goal

```
Type variables: <none>

x_: int
y_: int
--------------------------------------------
pre = M.x = x_ /\ M.y = y_

    M.f

post = M.y = x_ + y_
```

where the conclusion is just another way of writing the same Hoare triple.

We begin by applying the tactic `proc`, which inlines the code of `f`, transforming this goal into:

Proof of First Example

```
Type variables: <none>

x_: int
y_: int
--------------------------------------------
Context : M.f

pre = M.x = x_ /\ M.y = y_

(1----)  if (0 <= M.x) {
(1.1--)    while (0 < M.x) {
(1.1.1)      M.x <- M.x - 1
(1.1.2)      M.y <- M.y + 1
(1.1--)    }
(1----)  } else {
(1?1--)    while (M.x < 0) {
(1?1.1)      M.x <- M.x + 1
(1?1.2)      M.y <- M.y - 1
(1?1--)    }
(1----)  }

post = M.y = x_ + y_
```

Proof of First Example

Because the *first* statement is an `if`, we can use the tactic `if` to split this goal into two subgoals, depending upon whether `M.x` is non-negative or not:

```
Type variables: <none>

x_: int
y_: int
--------------------------------------------
Context : M.f

pre = (M.x = x_ /\ M.y = y_) /\ 0 <= M.x

(1--)  while (0 < M.x) {
(1.1)    M.x <- M.x - 1
(1.2)    M.y <- M.y + 1
(1--)  }

post = M.y = x_ + y_
```

(for the "then" part) and

Proof of First Example

```
Type variables: <none>

x_: int
y_: int
--------------------------------------------
Context : M.f

pre = (M.x = x_ /\ M.y = y_) /\ ! 0 <= M.x

(1--)  while (M.x < 0) {
(1.1)    M.x <- M.x + 1
(1.2)    M.y <- M.y - 1
(1--)  }

post = M.y = x_ + y_
```

(for the “else” part).

Proof of First Example

With both of these subgoals, the *final* (only in this case) statement is a while loop, and thus we can apply the `while` tactic, for which we need to supply an invariant. We'll only consider the proof of the first subgoal, the other being similar.

It's perhaps obvious that the invariant should include that the sum of `M.x` and `M.y` is equal to the sum of `x_` and `y_`. But we'll also need that `0 <= M.x`.

In the goal where `0 <= M.x`, running

```
while (0 <= M.x /\ M.x + M.y = x_ + y_).
```

generates the two subgoals

Proof of First Example

```
Type variables: <none>

x_: int
y_: int
--------------------------------------------
Context : M.f

pre =
  (0 <= M.x /\ M.x + M.y = x_ + y_) /\ 0 < M.x

(1)  M.x <- M.x - 1
(2)  M.y <- M.y + 1

post = 0 <= M.x /\ M.x + M.y = x_ + y_
```

(showing that the body of the loop preserves the invariant when `M.x` is positive) and

Proof of First Example

```
Type variables: <none>

x_: int
y_: int
--------------------------------------------
Context : M.f

pre = (M.x = x_ /\ M.y = y_) /\ 0 <= M.x


post =
  (0 <= M.x /\ M.x + M.y = x_ + y_) /\
  forall (x y : int),
    ! 0 < x =>
    0 <= x /\ x + y = x_ + y_ => y = x_ + y_
```

(connecting the while loop to the pre- and postconditions of the goal on which the `while` tactic was run). We'll come back to this second subgoal; but first, let's consider how to prove the first one.

Proof of First Example

To prove

```
Type variables: <none>

x_: int
y_: int
------------------------------------------
Context : M.f

pre =
  (0 <= M.x /\ M.x + M.y = x_ + y_) /\ 0 < M.x

(1)  M.x <- M.x - 1
(2)  M.y <- M.y + 1

post = 0 <= M.x /\ M.x + M.y = x_ + y_
```

we can push the assignments at the *end* of the program (all of the program in this case) into the postcondition, using the tactic `wp`, which stands for "weakest precondition".

Proof of First Example

In terms of the logic learned in class, it's equivalent to repeated use of the rule for assignment, combined with what the slides called the Rule of Hoare Logic Composition. This results in the goal:

```
Type variables: <none>

x_: int
y_: int
--------------------------------------------
Context : M.f

pre =
  (0 <= M.x /\ M.x + M.y = x_ + y_) /\ 0 < M.x


post =
  let x = M.x - 1 in
  0 <= x /\ x + (M.y + 1) = x_ + y_
```

Proof of First Example

Because the program of

```
Type variables: <none>

x_: int
y_: int
--------------------------------------------
Context : M.f

pre =
  (0 <= M.x /\ M.x + M.y = x_ + y_) /\ 0 < M.x


post =
  let x = M.x - 1 in
  0 <= x /\ x + (M.y + 1) = x_ + y_
```

is *empty*, we can use the `skip` tactic to reduce it to the ambient logic formula:

Proof of First Example

```
Type variables: <none>

x_: int
y_: int
--------------------------------------------
forall &hr,
  (0 <= M.x{hr} /\ M.x{hr} + M.y{hr} = x_ + y_) /\
  0 < M.x{hr} =>
  let x = M.x{hr} - 1 in
  0 <= x /\ x + (M.y{hr} + 1) = x_ + y_
```

Here `&hr` stands for an arbitrary memory, and `M.x{hr}` and `M.y{hr}` stand for the values of `M.x` and `M.y` in that memory. We can solve this goal by running the tactic `smt()`.

Proof of First Example

Now let's go back to the second subgoal generated by running the `while` tactic:

```
Type variables: <none>

x_: int
y_: int
--------------------------------------------
Context : M.f

pre = (M.x = x_ /\ M.y = y_) /\ 0 <= M.x


post =
  (0 <= M.x /\ M.x + M.y = x_ + y_) /\
  forall (x y : int),
    ! 0 < x =>
    0 <= x /\ x + y = x_ + y_ => y = x_ + y_
```

Here there is no program, because nothing came before the `while` loop.

Proof of First Example

The post condition

```
(0 <= M.x /\ M.x + M.y = x_ + y_) /\
forall (x y : int),
  ! 0 < x =>
  0 <= x /\ x + y = x_ + y_ => y = x_ + y_
```

has two conjuncts.

The first is the invariant specified to the `while` tactic, as it must be true that when the while loop is entered, the invariant holds.

Proof of First Example

Postcondition:

```
(0 <= M.x /\ M.x + M.y = x_ + y_) /\
forall (x y : int),
  ! 0 < x =>
  0 <= x /\ x + y = x_ + y_ => y = x_ + y_
```

The second part quantifies over the values `x` and `y`, representing the values of the variables modified by the while loop at the point where the loop is exited. It has implications assuming that the boolean expression of the while loop is false, and the loop's invariant holds, and requiring us to prove that the original postcondition (`M.y = x_ + y_`) holds—all expressed in terms of `x` and `y` instead of `M.x` and `M.y`.

The combination of `! 0 < x` and `0 <= x` tells us that `x` is zero, which is why `y = x_ + y_` holds, and also why `0 <= x` needed to be part of the invariant.

Proof of First Example

Because the goal's program part is empty, running `skip` reduces the goal to:

```
Type variables: <none>

x_: int
y_: int
--------------------------------------------
forall &hr,
  (M.x{hr} = x_ /\ M.y{hr} = y_) /\ 0 <= M.x{hr} =>
  (0 <= M.x{hr} /\ M.x{hr} + M.y{hr} = x_ + y_) /\
  forall (x y : int),
    ! 0 < x =>
    0 <= x /\ x + y = x_ + y_ => y = x_ + y_
```

And running `smt()` will solve this goal.

Proof of First Example

Note that only the variables *modified* by the while loop are universally quantified in the postcondition. Thus if the postcondition Φ of the goal on which the `while` tactic is run refers to variables used by the part of the program that comes before the while loop, or by the precondition of the goal on which the `while` tactic is run, whatever is known about those variables upon entry to the while loop can be used when proving Φ.

Second Example

Because procedures can take arguments and return results, here's an alternative version of our example:

```
module M’ = {
  proc f(x : int, y : int) : int = {
    var x’, y’ : int;
    x’ <- x; y’ <- y;
    if (0 <= x’) {
      while (0 < x’) {
        x’ <- x’ - 1; y’ <- y’ + 1;
      }
    }
    else {
      while (x’ < 0) {
        x’ <- x’ + 1; y’ <- y’ - 1;
      }
    }
    return y’;
  }
}.
```

Second Example

Here:

- `x` and `y` are arguments of `f`,
- the variables manipulated by the while loops are local variables `x’` and `y’`, and
- `y’` is explicitly returned as the result of `f`.

This time the lemma to be proved is:

```
lemma correct’ (x_ y_ : int) :
  hoare[M’.f : x = x_ /\ y = y_ ==> res = x_ + y_].
```

Note how the precondition refers to the values of `f`’s arguments, and how `res` in the postcondition is used to stand for the result returned by `f`.

Proof of Second Example

The proof of the second example is only slightly different from that of the first one. We start with the goal

```
Type variables: <none>

x_: int
y_: int
--------------------------------------------
pre = x = x_ /\ y = y_

    M'.f

post = res = x_ + y_
```

Running `proc` then gives us the goal

Proof of Second Example

```
Type variables: <none>

x_: int
y_: int
----------------------------------------------------------
Context : M'.f

pre = (x, y).`1 = x_ /\ (x, y).`2 = y_

(1----)  x' <- x
(2----)  y' <- y
(3----)  if (0 <= x') {
(3.1--)    while (0 < x') {
(3.1.1)      x' <- x' - 1
(3.1.2)      y' <- y' + 1
(3.1--)    }
(3----)  } else {
(3?1--)    while (x' < 0) {
(3?1.1)      x' <- x' + 1
(3?1.2)      y' <- y' - 1
(3?1--)    }
(3----)  }

post = y' = x_ + y_
```

Proof of Second Example

Note that the postcondition now involves `y'` not `res`, since `y'` is what is returned by `f`.

The precondition involves the notation for selecting the first or second component of a pair. If we run the tactic `simplify`, we get the goal:

Proof of Second Example

```
Type variables: <none>

x_: int
y_: int
-------------------------------------------
Context : M'.f

pre = x = x_ /\ y = y_

(1----)  x' <- x
(2----)  y' <- y
(3----)  if (0 <= x') {
(3.1--)    while (0 < x') {
(3.1.1)      x' <- x' - 1
(3.1.2)      y' <- y' + 1
(3.1--)    }
(3----)  } else {
(3?1--)    while (x' < 0) {
(3?1.1)      x' <- x' + 1
(3?1.2)      y' <- y' - 1
(3?1--)    }
(3----)  }

post = y' = x_ + y_
```

Proof of Second Example

Because the if statement is *not* the first statement of the program, we can't directly run the `if` tactic. Instead we must use EasyCrypt's sequencing tactic (based on the Rule of Hoare Logic Composition) to split this goal into one involving the first two assignments, and one involving the if statement.

We run the tactic

```
seq 2 : (x' = x_ /\ y' = y_).
```

Here the 2 is the number of statements to use for the first subgoal, and the condition will be used as the postcondition of the first subgoal, and the precondition of the second subgoal. Here are the goals we get after running this tactic:

Proof of Second Example

```
Type variables: <none>

x_: int
y_: int
--------------------------------------------
Context : M'.f

pre = x = x_ /\ y = y_

(1)  x' <- x
(2)  y' <- y

post = x' = x_ /\ y' = y_
```

(which we know how to solve using `wp; skip; trivial`) and

Proof of Second Example

```
Type variables: <none>

x_: int
y_: int
--------------------------------------------
Context : M'.f

pre = x' = x_ /\ y' = y_

(1----)  if (0 <= x') {
(1.1--)    while (0 < x') {
(1.1.1)      x' <- x' - 1
(1.1.2)      y' <- y' + 1
(1.1--)    }
(1----)  } else {
(1?1--)    while (x' < 0) {
(1?1.1)      x' <- x' + 1
(1?1.2)      y' <- y' - 1
(1?1--)    }
(1----)  }

post = y' = x_ + y_
```

(which is proved just like the analogous goal of the first example).

Proof of Second Example

Here is the complete proof of the second example:

```
lemma correct' (x_ y_ : int) :
  hoare[M'.f : x = x_ /\ y = y_ ==> res = x_ + y_].
proof.
proc; simplify.
seq 2 : (x' = x_ /\ y' = y_).
wp; skip; trivial.
if.
while (0 <= x' /\ x' + y' = x_ + y_).
wp; skip; smt().
skip; smt().
while (x' <= 0 /\ x' + y' = x_ + y_).
wp; skip; smt().
skip; smt().
qed.
```

More on `wp` *Tactic*

The `wp` tactic can actually push (possibly nested) conditionals and assignment statements at the end of the program into the postcondition. E.g., if the program is

```
module L = {
  var w : int

  proc f(x y : int) : unit = {
    if (x < y) {
      w <- y - x;
    }
    else {
      w <- x - y;
    }
  }
}.
```

then running

```
wp.
```

More on wp *Tactic*

transforms the goal

```
Type variables: <none>

--------------------------------------------
Context : L.f

pre = true

(1--)  if (x < y) {
(1.1)    L.w <- y - x
(1--)  } else {
(1?1)    L.w <- x - y
(1--)  }

post = 0 <= L.w
```

into

More on `wp` *Tactic*

transforms the goal

```
Type variables: <none>

------------------------------------------
Context : L.f

pre = true


post = if x < y then 0 <= y - x else 0 <= x - y
```