# EasyCrypt's *Relational Hoare Logic and Noninterference*

These slides are an example-based introduction to EasyCrypt's Relational Hoare Logic, focusing on how it can be used to prove noninterference results.

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

# First Example

Let's start with this simple program:

```
module M1 = {
  var x : int (* private *)
  var y : int (* public *)
  proc f() : unit = {
    x <- y;
  }
}.
```

EasyCrypt doesn't have a way of saying whether module variables or inputs/outputs to procedures should be considered to be "public" or "private", but in this and the subsequent examples, we'll note this using comments.

# *First Example*

We can state the noninterference lemma for

```
module M1 = {
  var x : int (* private *)
  var y : int (* public *)
  proc f() : unit = {
    x <- y;
  }
}.
```

as a Relational Hoare quadruple, as follows:

```
lemma lem1 :
  equiv [M1.f ~ M1.f : M1.y{1} = M1.y{2} ==> M1.y{1} = M1.y{2}].
```

In this notation, the two programs (identical, when stating noninterference), are separated by a tilde. They are followed by the pre- and postconditions, in which we use the notation `{1}` or `{2}` to say which memory we want a variable or expression to be interpreted in.

## *First Example*

So in

```
lemma lem1 :
  equiv [M1.f ~ M1.f : M1.y{1} = M1.y{2} ==> M1.y{1} = M1.y{2}].
```

we are saying that if the values of the public variable `M1.y` in the two memories are equal before running `M1.f`, that either both executions of `M1.f` fail to terminate (which does not happen in this case), or they both terminate, and the values of `M1.y` in the resulting memories are equal.

## *First Example*

When we prove this lemma, we are initially presented with the goal

```
Type variables: <none>

------------------------------------------------------------------------
pre = ={M1.y}

    M1.f ~ M1.f

post = ={M1.y}
```

Note that `M1.y{1} = M1.y{2}` has been abbreviated to `={M1.y}`. We can use such abbreviations ourselves, writing, e.g.,

```
={x, y}
```

instead of

```
x{1} = x{2} /\ y{1} = y{2}.
```

This only works with variables, not expressions.

## First Example

As in Hoare logic, we start by running the tactic

```
proc.
```

to transform our goal

```
Type variables: <none>

------------------------------------------------------------------------
pre = ={M1.y}

    M1.f ~ M1.f

post = ={M1.y}
```

into

# *First Example*

```
Type variables: <none>

--------------------------------------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre = ={M1.y}

(1)  M1.x <- M1.y

post = ={M1.y}
```

Here the programs are in sync, and so are only listed once. `&1` and `&2` are how the memories of the two programs are named. For this goal, we can run

```
wp.
```

which in Relational Hoare Logic pushes the possibly nested conditionals and assignments at the *ends* of the two programs into the postcondition, giving us the goal

## First Example

```
Type variables: <none>

-------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre = ={M1.y}


post = ={M1.y}
```

Note that the postcondition did not change, because there were no ocurrences of the left-hand-side of the assignment in the postcondition.

From here, just as in Hoare Logic, we can run

```
skip.
```

which gives us the goal

# First Example

```
Type variables: <none>

----------------------------------------------------------------------
forall &1 &2, ={M1.y} => ={M1.y}
```

The conclusion of this goal is the Ambient Logic formula saying that for all memories `&1` (of the first program) and `&2` (of the second program), that if the values of `M1.y` in the two memories (`M1.y{1}` and `M1.y{2}`) are equal, that the values of `M1.y` in the two memories are equal. This can be proved by running

```
trivial.
```

## *First Example*

Just as in Hoare Logic, we can abbreviate the proof of our lemma to

```
lemma lem1 :
  equiv [M1.f ~ M1.f : M1.y{1} = M1.y{2} ==> M1.y{1} = M1.y{2}].
proof.
proc; wp; skip; trivial.
qed.
```

And also like in Hoare Logic, the tactic `auto` tries to use `wp`, `skip` and `trivial` to solve a goal, and we can in fact abbreviate our proof to

```
lemma lem1 :
  equiv [M1.f ~ M1.f : M1.y{1} = M1.y{2} ==> M1.y{1} = M1.y{2}].
proof.
proc; auto.
qed.
```

## *Second Example*

For our second example, consider the program

```
module M2 = {
  var x : int (* private *)
  var y : int (* public *)
  proc f() : unit = {
    y <- x;
  }
}.
```

Here we've swapped `x` and `y` in the assignment, so if we try to prove

```
lemma lem2 :
  equiv [M2.f ~ M2.f : ={M2.y} ==> ={M2.y}].
proof.
proc; wp; skip.
```

we are given the goal

## *Second Example*

```
Type variables: <none>

----------------------------------------------
&1 (left ) : {}
&2 (right) : {}

pre = Z.b{1} /\ !Z.b{2}

if (Z.b) {                (1--)  if (Z.b) {
  Z.x <- 1                (1.1)    Z.x <- 2
} else {                  (1--)  } else {
  Z.x <- 2                (1?1)    Z.x <- 1
}                         (1--)  }

post = ={Z.x}
```

This goal cannot be solved, as knowing that the values in the two memories of `M2.y` are equal is of no help in concluding that the values in the two memories of `M2.x` are equal.

We can thus run

```
abort.
```

# *Third Example*

Consider the following program and proof beginning

```
module M3 = {
  var x : int (* private *)
  var y : int (* public *)
  proc f() : unit = {
    y <- x;
    y <- 5;
  }
}.

lemma lem3 :
  equiv [M3.f ~ M3.f : ={M3.y} ==> ={M3.y}].
proof.
proc.
```

which take us to the goal

# *Third Example*

```
Type variables: <none>

--------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre = ={M3.y}

(1)  M3.y <- M3.x
(2)  M3.y <- 5

post = ={M3.y}
```

Running

```
wp.
```

then takes us to the goal

## *Third Example*

```
Type variables: <none>

--------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre = ={M3.y}


post = 5 = 5
```

(Note how the first assignment has no effect on the postcondition, because `wp` applied it after `M3.y{1}` and `M3.y{2}` had been replaced by 5.) This goal can be solved by running

```
auto.
```

## Fourth Example

As our fourth example, consider the program and proof beginning

```
module M4 = {
  var x : int (* private *)
  var y : int (* public *)
  proc f() : unit = {
    if (y %% 3 = 0) {
      x <- 0;
    }
    else {
      x <- 1;
    }
  }
}.
```

and

```
lemma lem4 :
  equiv [M4.f ~ M4.f : ={M4.y} ==> ={M4.y}].
proof.
proc.
```

which takes us to the goal

# *Fourth Example*

```
Type variables: <none>

-------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre = ={M4.y}

(1--)  if (M4.y %% 3 = 0) {
(1.1)    M4.x <- 0
(1--)  } else {
(1?1)    M4.x <- 1
(1--)  }

post = ={M4.y}
```

Because *both* programs *begin* with conditionals (equal in our case), we can apply the two-sided `if` tactic

```
if.
```

which gives us three subgoals

## *Fourth Example*

```
Type variables: <none>

--------------------------------------------
forall &1 &2,
  ={M4.y} =>
  M4.y{1} %% 3 = 0 <=> M4.y{2} %% 3 = 0
```

(which makes us prove that the boolean expression of the first program's conditional holds in the first program's memory if-and-only-if the boolean expression of the second program's conditional holds in the second program's memory; in our case, the conditionals and so their boolean expressions are the same, of course) and

# *Fourth Example*

```
Type variables: <none>

-------------------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre = ={M4.y} /\ M4.y{1} %% 3 = 0

(1)  M4.x <- 0

post = ={M4.y}
```

(for the then branch—if the conditionals of the two programs were different, we'd have the then branch of the first conditional on the left, and the then branch of the second conditional on the right, followed in each case by whatever came after the conditional in the two programs) and

## *Fourth Example*

```
Type variables: <none>

--------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre = ={M4.y} /\ M4.y{1} %% 3 <> 0

(1)  M4.x <- 1

post = ={M4.y}
```

(for the `else` branch—again, if the programs were not synchronized we'd have a pair of else branches, followed by whatever followed in the two programs). The second and third subgoals follow easily because `M4.x` does not appear in the postconditions (which are equal).

# *Fifth Example*

On the other hand, suppose we modify the previous example so that we branch on whether the private variable `x` is divisible by 3, and set the public variable `y` instead of `x`:

```
module M5 = {
  var x : int (* private *)
  var y : int (* public *)
  proc f() : unit = {
    if (x %% 3 = 0) {
      y <- 0;
    }
    else {
      y <- 1;
    }
  }
}.
```

# *Fifth Example*

Then, the proof beginning

```
lemma lem5 :
  equiv [M5.f ~ M5.f : ={M5.y} ==> ={M5.y}].
proof.
proc; if.
```

takes us to the three subgoals

```
Type variables: <none>

--------------------------------------------
forall &1 &2,
  ={M5.y} =>
  M5.x{1} %% 3 = 0 <=> M5.x{2} %% 3 = 0
```

and

# *Fifth Example*

```
Type variables: <none>

---------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre = ={M5.y} /\ M5.x{1} %% 3 = 0

(1)  M5.y <- 0

post = ={M5.y}
```

and

# *Fifth Example*

```
Type variables: <none>

--------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre = ={M5.y} /\ M5.x{1} %% 3 <> 0

(1)  M5.y <- 1

post = ={M5.y}
```

Because we don't know that the values of the private `M5.x` in the two memories are related in any way, we can't complete this proof. (There is a one-sided `if` tactic, which we'll see in the next example. But it won't help either.)

# *Sixth Example*

For our sixth example, consider the program

```
require import List.

module M6 = {
  var i : int        (* public *)
  var xs : int list (* public *)
  var ys : int list (* private *)
  var r : bool       (* private *)

  proc f() : unit = {
    i <- 0;
    r <- false;
    while (i < 10) {
      if (! (nth 0 xs i = nth 1 ys i)) {
        r <- true;
      }
      i <- i + 1;
    }
  }
}.
```

# *Sixth Example*

Here we have imported the theory `List` from the EasyCrypt Library, so that the type `int list` consists of all finite lists of integers. We do list subscripting using the operator `nth`: `nth def xs i`,

- returns the `i`th (counting from `0`) element of `xs`, if `i` is at least 0 and is strictly less than the number of elements in `xs`; and
- returns the default element `def`, otherwise.

For example:

- the value of `nth 6 [1; 2; 3] 1` is 2;
- the value of `nth 6 [1; 2; 3] (-1)` is 6;
- the value of `nth 6 [1; 2; 3] 3` is 6.

## *Sixth Example*

Because the default values supplied to `nth` in

```
while (i < 10) {
  if (! (nth 0 xs i = nth 1 ys i)) {
    r <- true;
  }
  i <- i + 1;
}
```

are different but might also appear in the lists, `r` can be set to `true` for the first time because

- we reach a point where `i` is a good index for both `xs` and `ys`, but the `i`th elements of `xs` and `ys` are different;
- we reach a point where `i` is a bad index for both `xs` and `ys`;
- we reach a point where `i` is a good index for `xs` and a bad index for `ys`, but the `i`th element of `xs` is not 1;
- we reach a point where `i` is a bad index for `xs` and a good index for `ys`, but the `i`th element of `ys` is not 0.

## *Sixth Example*

Let's prove the lemma

```
lemma lem6 :
  equiv [M6.f ~ M6.f : ={M6.i, M6.xs} ==> ={M6.i, M6.xs} /\ P].
```

where the operator `P` is defined by

```
op P (x : bool * bool) : bool = true.
```

and is only included in the postcondition so as to help illustrate how the `while` tactic works. After running

```
proc.
```

we are at goal

# Sixth Example

```
Type variables: <none>

----------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre = ={M6.i, M6.xs}

(1----)  M6.i <- 0
(2----)  M6.r <- false
(3----)  while (M6.i < 10) {
(3.1--)    if (nth 0 M6.xs
(    -)          M6.i <>
(    -)        nth 1 M6.ys
(    -)          M6.i) {
(3.1.1)      M6.r <- true
(3.1--)    }
(3.2--)    M6.i <- M6.i + 1
(3----)  }

post = ={M6.i, M6.xs} /\ P (M6.r{1}, M6.r{2})
```

# *Sixth Example*

It's then convenient (but not necessary) to use the two-sided version of the `seq` tactic, which takes two arguments: the number of statements to take from the beginning of the left and right programs, respectively.

E.g., running

```
seq 2 2 : (={M6.i, M6.xs}).
auto.
```

takes us to the goal

# *Sixth Example*

```
Type variables: <none>

--------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre = ={M6.i, M6.xs}

(1----)  while (M6.i < 10) {
(1.1--)    if (nth 0 M6.xs
(    -)          M6.i <>
(    -)        nth 1 M6.ys
(    -)          M6.i) {
(1.1.1)      M6.r <- true
(1.1--)    }
(1.2--)    M6.i <- M6.i + 1
(1----)  }

post = ={M6.i, M6.xs} /\ P (M6.r{1}, M6.r{2})
```

# *Sixth Example*

Because *both* programs (they are in sync) *end* with `while` loops, we can apply the `while` tactic, choosing a loop invariant

```
while (={M6.i, M6.xs}).
```

saying that the values of the public variables `M6.i` and `M6.xs` stay equal in the two memories. This gives us the subgoals

# *Sixth Example*

```
Type variables: <none>

--------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre =
  ={M6.i, M6.xs} /\ M6.i{1} < 10 /\ M6.i{2} < 10

(1--)  if (nth 0 M6.xs
(  -)        M6.i <>
(  -)      nth 1 M6.ys
(  -)        M6.i) {
(1.1)    M6.r <- true
(1--)  }
(2--)  M6.i <- M6.i + 1

post =
  ={M6.i, M6.xs} /\
  (M6.i{1} < 10 <=> M6.i{2} < 10)
```

(goal 1—preservation of loop invariant) and

# *Sixth Example*

```
Type variables: <none>

----------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre = ={M6.i, M6.xs}


post =
  (={M6.i, M6.xs} /\
   (M6.i{1} < 10 <=> M6.i{2} < 10)) /\
  forall (i_L : int) (r_L : bool) (i_R : int)
    (r_R : bool),
    ! i_L < 10 =>
    ! i_R < 10 =>
    i_L = i_R /\ ={M6.xs} =>
    (i_L = i_R /\ ={M6.xs}) /\ P (r_L, r_R)
```

(goal 2—connection of loop with pre- and postconditions).

# *Sixth Example*

Let's consider goal 2, first. After we run

```
skip.
```

we have the goal

```
Type variables: <none>

--------------------------------------------
forall &1 &2,
  ={M6.i, M6.xs} =>
  (={M6.i, M6.xs} /\
   (M6.i{1} < 10 <=> M6.i{2} < 10)) /\
  forall (i_L : int) (r_L : bool) (i_R : int)
    (r_R : bool),
    ! i_L < 10 =>
    ! i_R < 10 =>
    i_L = i_R /\ ={M6.xs} =>
    (i_L = i_R /\ ={M6.xs}) /\ P (r_L, r_R)
```

## *Sixth Example*

The conclusion of this goal makes us prove two conjuncts, given the knowledge that the loop's precondition holds on the two memories. The first conjunct is

```
(={M6.i, M6.xs} /\ (M6.i{1} < 10 <=> M6.i{2} < 10))
```

In words, we have to show that the loop invariant is true at the beginning of the loop's execution, and that the boolean expression `M6.i < 10` is either true in both memories or false in both memories.

# *Sixth Example*

The second conjunct is

```
forall (i_L : int) (r_L : bool) (i_R : int) (r_R : bool),
  ! i_L < 10 => ! i_R < 10 =>
  i_L = i_R /\ ={M6.xs} =>
  (i_L = i_R /\ ={M6.xs}) /\ P (r_L, r_R)
```

It quantifies over the variables that *change* during the execution of the loop:

- `M6.i{1}`, which is turned into `i_L`;
- `M6.i{2}`, which is turned into `i_R`;
- `M6.r{1}`, which is turned into `r_L`; and
- `M6.r{2}`, which is turned into `r_R`.

(If we'd left out the conjunct `P (r{1}, r{2})` from the overall postcondition, EasyCrypt would have simplified away the entire second conjunct, making it easier to prove but harder to understand!)

## *Sixth Example*

When proving this second conjunct, we are given the knowledge that the boolean expression of the loop is `false` in both memories, but that the loop invariant holds. We then have to prove the postcondition of the loop.

## *Sixth Example*

Now, let's go back to the first subgoal:

```
Type variables: <none>

--------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre =
  ={M6.i, M6.xs} /\ M6.i{1} < 10 /\ M6.i{2} < 10

(1--)  if (nth 0 M6.xs
(  -)        M6.i <>
(  -)      nth 1 M6.ys
(  -)        M6.i) {
(1.1)    M6.r <- true
(1--)  }
(2--)  M6.i <- M6.i + 1

post =
  ={M6.i, M6.xs} /\
  (M6.i{1} < 10 <=> M6.i{2} < 10)
```

# *Sixth Example*

The pre- and postconditions both include the loop invariant.

In addition, the precondition tells us that the boolean expression holds in both memories (if the left and right programs were different while loops, we'd have that the left loop's boolean expression held in the first memory, and the right loop's boolean expression held in the second memory).

In the postcondition, we also have to prove that the left loop's boolean expression holds in the first memory if-and-only-if the right loop's boolean expression holds in the second memory.

# *Sixth Example*

Because the boolean expression of the conditional depends upon the possibly different values of the private variable `M6.ys` in the two memories, we can't use the two-sided `if` tactic. Instead we have to use its one-sided versions, which are applicable when the given program (one/left or two/right) *begins* with a conditional. Running

```
if{1}.
```

give us two subgoals where the second (right) program is unchanged. In the first subgoal, we are given the additional assumption (just about memory one) that

```
nth 0 M6.xs{1} M6.i{1} <> nth 1 M6.ys{1} M6.i{1}
```

and the left program becomes

```
M6.r <- true;          (* the then branch *)
M6.i <- M6.i + 1;      (* what follows the conditional *)
```

# *Sixth Example*

In the second subgoal, we are given the additional assumption (again about memory one) that

```
! (nth 0 M6.xs{1} M6.i{1} <> nth 1 M6.ys{1} M6.i{1})
```

and the left program becomes

```
                      (* the else branch - empty! *)
M6.i <- M6.i + 1;     (* what follows the conditional *)
```

In both of these subgoals, we must run the one-sided `if` tactic on the right program (program two)

```
if{2}.
```

All four of the resulting goals can then be solved using `auto`.

# *Sixth Example*

For example, the third of these goals is (some of what EasyCrypt prints has been elided so it fits on the slide!):

```
pre =
  ((={M6.i, M6.xs} /\ M6.i{1} < 10 /\ M6.i{2} < 10) /\
   ! nth 0 M6.xs{1} M6.i{1} <> nth 1 M6.ys{1} M6.i{1}) /\
  nth 0 M6.xs{2} M6.i{2} <> nth 1 M6.ys{2} M6.i{2}

M6.i <-                     (1)  M6.r <-
  M6.i +                    ( )    true
  1                         ( )
                            (2)  M6.i <-
                            ( )    M6.i +
                            ( )    1

post = ={M6.i, M6.xs} /\ (M6.i{1} < 10 <=> M6.i{2} < 10)
```

Here we have the `else` (empty) branch of the conditional of the left program, but the then branch of the conditional of the right program—because we're in the goal where the boolean expression was `false` in the first memory, but `true` in the second memory.

## Sixth Example

Going back again to the goal

```
Type variables: <none>

------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre =
  ={M6.i, M6.xs} /\ M6.i{1} < 10 /\ M6.i{2} < 10

(1--)  if (nth 0 M6.xs
(  -)        M6.i <>
(  -)      nth 1 M6.ys
(  -)        M6.i) {
(1.1)    M6.r <- true
(1--)  }
(2--)  M6.i <- M6.i + 1

post =
  ={M6.i, M6.xs} /\
  (M6.i{1} < 10 <=> M6.i{2} < 10)
```

## *Sixth Example*

it's worth noting that in Relational Hoare Logic, `wp` is capable of pushing possibly nested conditionals and assignments at the ends of the two programs into the postcondition. Running

```
wp.
```

transforms our goal into a goal with postcondition

## Sixth Example

```
if nth 0 M6.xs{2} M6.i{2} <> nth 1 M6.ys{2} M6.i{2} then
  let i_R = M6.i{2} + 1 in
  (if nth 0 M6.xs{1} M6.i{1} <> nth 1 M6.ys{1} M6.i{1} then
     let i_L = M6.i{1} + 1 in
     (i_L = i_R /\ ={M6.xs}) /\ (i_L < 10 <=> i_R < 10)
   else
     let i_L = M6.i{1} + 1 in
     (i_L = i_R /\ ={M6.xs}) /\ (i_L < 10 <=> i_R < 10))
else
  let i_R = M6.i{2} + 1 in
  (if nth 0 M6.xs{1} M6.i{1} <> nth 1 M6.ys{1} M6.i{1} then
     let i_L = M6.i{1} + 1 in
     (i_L = i_R /\ ={M6.xs}) /\ (i_L < 10 <=> i_R < 10)
   else
     let i_L = M6.i{1} + 1 in
     (i_L = i_R /\ ={M6.xs}) /\ (i_L < 10 <=> i_R < 10))
```

This goal can be solved with

```
skip; trivial.
```

so we could actually solve the original goal with auto.

# *Sixth Example*

If we only want to prove noninterference, we can get rid of the use of `P` in the postcondition:

```
lemma lem :
  equiv [M6.f ~ M6.f : ={M6.i, M6.xs} ==> ={M6.i, M6.xs}].
```

Furthermore, because our program ends with a `while` loop, and the proof of the first subgoal generated by the `while` tactic doesn't actually depend on `xs` being the same in the two memories, we can begin our proof like this:

```
proc.
while (={M6.i}).
```

This gives us the goals

# *Sixth Example*

```
Type variables: <none>

--------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre = ={M6.i} /\ M6.i{1} < 10 /\ M6.i{2} < 10

(1--)  if (nth 0 M6.xs
(  -)        M6.i <>
(  -)      nth 1 M6.ys
(  -)        M6.i) {
(1.1)    M6.r <- true
(1--)  }
(2--)  M6.i <- M6.i + 1

post = ={M6.i} /\ (M6.i{1} < 10 <=> M6.i{2} < 10)
```

(which can be solved with auto) and

# *Sixth Example*

```
Type variables: <none>

--------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre = ={M6.i, M6.xs}

(1)  M6.i <- 0
(2)  M6.r <- false

post =
  (={M6.i} /\ (M6.i{1} < 10 <=> M6.i{2} < 10)) /\
  forall (i_L i_R : int),
    ! i_L < 10 =>
    ! i_R < 10 =>
    i_L = i_R => i_L = i_R /\ ={M6.xs}
```

(which can also be solved by `auto`, because the occurrence of `={M6.xs}` in the postcondition is assumed in the precondition).

## *Sixth Example*

Thus our lemma and its proof can be:

```
lemma lem :
  equiv [M6.f ~ M6.f : ={M6.i, M6.xs} ==> ={M6.i, M6.xs}].
proof.
proc; while (={M6.i}); auto.
qed.
```

# *Seventh Example*

Now, let's take our sixth example and restructure it so

- the lists `xs` (public) and `ys` (private) are arguments to the procedure `M7.f`; and
- the variables that are initialized without reference to the arguments—`i` (public) and `r` (private)—are returned as the procedure's result;

Because neither `xs` nor `ys` are modified, we don't return them.

## *Seventh Example*

So our program is now

```
module M7 = {
  proc f(xs : int list,   (* public *)
         ys : int list)   (* private *)
         : int *          (* i's value - public *)
           bool = {       (* r's value - private *)
    var i : int;          (* public *)
    var r : bool;         (* private *)
    i <- 0;
    r <- false;
    while (i < 10) {
      if (! (nth 0 xs i = nth 1 ys i)) {
        r <- true;
      }
      i <- i + 1;
    }
    return (i, r);
  }
}.
```

# *Seventh Example*

And our noninterference lemma and proof are:

```
lemma lem7 :
  equiv [M7.f ~ M7.f : ={xs} ==> res{1}.`1 = res{2}.`1].
  (* the second character of .` is the backtick character *)
proof.
proc; while (={i}); auto.
qed.
```

## *The* `sp` *tactic and Optional Arguments to* `wp` *and* `sp`

Like in Hoare Logic, the `sp` tactic in Relational Hoare Logic will push possibly nested conditionals involving assignments at the beginning of both programs into the precondition.

Furthermore, both `wp` and `sp` optionally take a pair of natural number arguments, pertaining to the left and right programs, respectively. E.g.,

```
sp 1 2.
```

will try to push the first statement of the left program and the first two statements of the right program into the precondition, failing if either is not possible.

# *The* `case` *and* `exfalso` *Tactics*

As with Hoare Logic, Relational Hoare Logic has the `case` and `exfalso` tactics, which work analogously, except that their arguments are now formulas that can mention variables of both memories.

## *The* `rcondt` *and* `rcondf` *Tactics*

Furthermore, Relational Hoare Logic also has the `rcondt` and `rcondf` tactics, which apply to both conditionals and while loops. But unlike in Hoare Logic, these tactics must be annotated with the side (left or right program) they should be applied to.

# *The* `rcondt` *and* `rcondf` *Tactics*

For example, consider the module

```
module Z = {
  var b : bool
  var x : int
  proc f() : unit = {
    if (b) {
      x <- 1;
    }
    else {
      x <- 2;
    }
  }
  proc g() : unit = {
    if (b) {
      x <- 2;
    }
    else {
      x <- 1;
    }
  }
}.
```

## *The* rcondt *and* rcondf *Tactics*

We can start proof of the following lemma as indicated:

```
lemma Z :
  equiv [Z.f ~ Z.g : Z.b{1} /\ ! Z.b{2} ==> Z.x{1} = Z.x{2}].
proof.
proc.
```

This gives us the goal:

## *The* `rcondt` *and* `rcondf` *Tactics*

```
Type variables: <none>

-------------------------------------------
&1 (left ) : {}
&2 (right) : {}

pre = Z.b{1} /\ !Z.b{2}

if (Z.b) {                (1--)  if (Z.b) {
  Z.x <- 1                (1.1)    Z.x <- 2
} else {                  (1--)  } else {
  Z.x <- 2                (1?1)    Z.x <- 1
}                         (1--)  }

post = ={Z.x}
```

Then, running the tactic

```
rcondt{1} 1.
```

gives us two goals.

## *The* rcondt *and* rcondf *Tactics*

The first is

```
Type variables: <none>

------------------------------------------
forall &m,
  hoare[ <skip> : Z.b /\ !Z.b{m} ==> Z.b]
```

where `&m` stands for the memory of the second program, and the `<skip>` indicates that there were no instructions before the indicated conditional.
Running the tactic

```
move => &m.
```

gives us the goal

# *The* rcondt *and* rcondf *Tactics*

```
Type variables: <none>

&m: {}
---------------------------------------------
Context : {}

pre = Z.b /\ !Z.b{m}


post = Z.b
```

which can be solved by auto.

## *The* rcondt *and* rcondf *Tactics*

The second goal produced by rcondt is:

```
Type variables: <none>

------------------------------------------
&1 (left ) : {}
&2 (right) : {}

pre = Z.b{1} /\ !Z.b{2}

Z.x <- 1                      (1--)  if (Z.b) {
                              (1.1)    Z.x <- 2
                              (1--)  } else {
                              (1?1)    Z.x <- 1
                              (1--)  }

post = ={Z.x}
```

Then, running the tactic

```
rcondf{2} 1.
```

gives us the goals

## *The* rcondt *and* rcondf *Tactics*

```
Type variables: <none>

-------------------------------------------
forall &m,
  hoare[ <skip> : Z.b{m} /\ !Z.b ==> !Z.b]
```

(which can be solved by auto) and

## *The* rcondt *and* rcondf *Tactics*

```
Type variables: <none>

---------------------------------------------
&1 (left ) : {} [programs are in sync]
&2 (right) : {}

pre = Z.b{1} /\ !Z.b{2}

(1)  Z.x <- 1

post = ={Z.x}
```

(which can also be solved by `auto`).