# *Using EasyCrypt's Probabilistic Hoare Logic, Probabilistic Relational Hoare Logic and ambient logic in Conjunction*

These slides are an example-based introduction to EasyCrypt's Probabilistic Hoare Logic (pHL) and how this logic can be used in conjunction with EasyCrypt's Probabilistic Relational Hoare Logic (pRHL) and ambient logic.

phL lets us bound the probability that running a procedure terminates in a memory satifying some event (predicate).

Many of the tactics of pHL work similarly to those of EasyCrypt's Hoare Logic (experiment!), and so we'll focus on the differences.

# *Examples*

We start our examples (see `phl-prhl.ec`) with

```
require import AllCore Distr DBool StdOrder.
import RealOrder.
```

`Distr` has definitions and lemmas about sub-distributions, `DBool` has the distribution `{0,1}` on `bool` that assigns both `true` and `false` weight one-half, `StdOrder` has lemmas about orderings (`<`, `<=`), and we import its sub-theory `RealOrder` which has lemmas about orderings on `real`.

The name `{0,1}` is misleading, as the elements of type `bool` are not `0` and `1`, and it suggests one can also write, e.g., `{1,0}`, which is not correct.

## First Example

Our first example is concerned with the modules

```
module M = {
  proc f() : bool = {
    var b : bool;
    b <$ {0,1};
    return b;
  }
}.
module N = {
  proc f() : bool = {
    var b1, b2 : bool;
    b1 <$ {0,1}; b2 <$ {0,1};
    return b1 ^ b2;  (* exclusive or *)
  }
}.
```

`M.f` returns a random boolean, whereas `N.f` returns the exclusive or of two random booleans.

# *First Example*

Using the approach we have already studied, one can prove this pRHL judgement:

```
lemma M_N_equiv :
  equiv [M.f ~ N.f : true ==> ={res}].
proof.
proc.
seq 0 1 : true.
rnd{2}.
auto.
rnd (fun x => x ^ b1{2}).
auto; smt().
qed.
```

Note that EasyCrypt automatically recognizes that {0,1} is lossless, and so this proof doesn't have to explicitly invoke the lemma `dbool_ll` from `DBool`.

# *First Example*

Now we can use `M_N_equiv` to prove that, no matter what memory, `&m`, they are started in (`M` and `N` have no global variables, so using different memories will have the same effect), `M.f` and `N.f` are equally likely to return `true`.

```
lemma M_N_true &m :
  Pr[M.f() @ &m : res] = Pr[N.f() @ &m : res].
```

This form is recognized by the ambient logic tactic `byequiv`.

# *First Example*

Running the tactic

```
byequiv (_ : true ==> ={res}).
```

in goal

```
Type variables: <none>

&m: {}
--------------------------------------------
Pr[M.f() @ &m : res] = Pr[N.f() @ &m : res]
```

results in three subgoals, the first of which is

# First Example

```
Type variables: <none>

&m: {}
--------------------------------------------
pre = true

    M.f ~ N.f

post = ={res}
```

and which we can solve with

```
apply M_N_equiv.
```

The second subgoal can be solved with `trivial`, as it makes us prove that the precondition of this pRHL judgement is established. And the third subgoal is

# *First Example*

```
Type variables: <none>

&m: {}
---------------------------------------------
forall &1 &2, ={res} => res{1} <=> res{2}
```

This makes us prove that the postcondition of the pRHL judgement implies that the two events, `res`, of

```
Pr[M.f() @ &m : res] = Pr[N.f() @ &m : res]
```

are in an iff relationship in their respective memories. We can also solve this goal using `trivial`.

# *First Example*

If we use `byequiv` with no argument, it defaults to the argument we supplied, and we can also explicitly tell it which lemma to use:

```
lemma M_N_true' &m :
  Pr[M.f() @ &m : res] = Pr[N.f() @ &m : res].
proof.
byequiv => //.
apply M_N_equiv.
qed.

lemma M_N_true'' &m :
  Pr[M.f() @ &m : res] = Pr[N.f() @ &m : res].
proof.
by byequiv M_N_equiv.
qed.
```

## *First Example*

Furthermore, we can use the same approach to prove that `M.f` and `N.f` are equally likely to return `false`:

```
lemma M_N_false &m :
  Pr[M.f() @ &m : !res] = Pr[N.f() @ &m : !res].
proof.
by byequiv M_N_equiv.
qed.
```

# *First Example*

Next, we might want to prove that `M.f` returns `true` exactly half the time:

```
lemma M_true &m :
  Pr[M.f() @ &m : res] = 1%r / 2%r.
```

The conclusion of this goal can be handled by the `byphoare` ambient logic tactic.

Running

```
byphoare (_ : true ==> res)
```

transforms the goal

```
Type variables: <none>

&m: {}
--------------------------------------------
Pr[M.f() @ &m : res] = 1%r / 2%r
```

# *First Example*

into three sub-goals, the first of which is

```
Type variables: <none>

&m: {}
--------------------------------------------
pre = true

    M.f
    [=] 1%r / 2%r

post = res
```

The conclusion of this goal is a pHL judgement. This looks like a Hoare Logic judgement, except there is also a bound—in this case equality with `1%r / 2%r`. The meaning of this judgement is that running `M.f` in a memory satisfying the precondition `true` (and so any memory) terminates in a memory in which the result (`res`) is `true` exactly half the time.

# *First Example*

The second subgoal is to show that the precondition of this pHL judgement is established, and the third is so show that its postcondition is equivalent to the event (`res`) of the Pr formula. Both can be solved by `trivial`.

To solve the first subgoal, we first run `proc`, giving us the subgoal

```
Type variables: <none>

&m: {}
---------------------------------------------
Context : {b : bool}
Bound   : [=] 1%r / 2%r

pre = true

(1)  b <$ {0,1}

post = b
```

# *First Example*

Next, we need to push this random assignment into the postcondition using the `rnd` tactic, which takes an optional argument, a predicate on the boolean being sampled. To figure out the appropriate predicate, try starting with the predicate `pred0`, which is true of no booleans. In this case, the correct argument is

```
rnd (pred1 true).
```

Running this, gives us the goal

# *First Example*

```
Type variables: <none>

&m: {}
--------------------------------------------
Context : {b : bool}
Bound   : [=] 1%r

pre = true /\ true


post =
  mu1 {0,1} true = 1%r / 2%r &&
  forall (v : bool),
    v \in {0,1} => pred1 true v <=> v
```

Note that the residual bound is now `[=] 1%r`. We can solve this goal by running

```
skip; progress.
smt(dbool1E). smt(). smt().
```

# *First Example*

Here is a more succinct version of `M_true`:

```
lemma M_true' &m :
  Pr[M.f() @ &m : res] = 1%r / 2%r.
proof.
byphoare => //.
proc.
rnd (pred1 true).
auto; smt(dbool1E).
qed.
```

And then we can prove:

```
lemma N_true &m :
  Pr[N.f() @ &m : res] = 1%r / 2%r.
proof.
by rewrite -(M_N_true &m) (M_true &m).
qed.
```

## First Example

Alternatively, we can prove `N_true` directly. We start with

```
lemma N_true' &m :
  Pr[N.f() @ &m : res] = 1%r / 2%r.
proof.
byphoare => //.
proc.
```

which takes us to the goal

```
Type variables: <none>

&m: {}
--------------------------------------------
Context : {b1, b2 : bool}
Bound   : [=] 1%r / 2%r

pre = true

(1)  b1 <$ {0,1}
(2)  b2 <$ {0,1}

post = b1 ^ b2
```

# *First Example*

To continue, we want to use the `seq` tactic to split the program after the first random assignment. In pHL, this tactic takes four additional arguments in comparison to the version of Hoare Logic. The defaults for these additional arguments do not work for our purposes.

We will run

```
seq 1 :
  b1             (* intermediate condition (IC) *)
  (1%r / 2%r)    (* (a) *)
  (1%r / 2%r)    (* (b) *)
  (1%r / 2%r)    (* (c) *)
  (1%r / 2%r).   (* (d) *)
```

# *First Example*

Here our intermediate condition (IC) is that `b1` holds, i.e., `b1` was assigned `true`. The next four arguments are probabilities, which we've labeled (a)–(d). (a) is the probability that if we run the first random assignment starting from a memory satisfying the precondition (`true`), that we'll terminate in a memory satisfying IC. (b) is the probability that running the second random assignment from a memory satisfying IC will terminate in a memory satisfing the postcondition `b1 ^ b2`. (c) is like (a), except it's for when the resulting memory satisfies the negation of IC. And (d) is like (b), except it's for when the starting point for running the second random assignment is a memory satisfying the negation of IC.

# *First Example*

Running the above `seq` gives us six subgoals. The conclusion of the first subgoal is a Hoare Logic judgement with postcondition `true`, and can thus be solved with `auto`. This first subgoal is only non-trivial if yet another optional argument is supplied to `seq`.

# First Example

The second subgoal is:

```
Type variables: <none>

&m: {}
------------------------------------------
Context : {b1, b2 : bool}
Bound   : [=] 1%r / 2%r

pre = true

(1)  b1 <$ {0,1}

post = b1
```

Here the bound is (a), and we can solve this goal with

```
rnd (pred1 true).
skip; progress.
smt(dbool1E). smt(). smt().
```

# *First Example*

The third subgoal is:

```
Type variables: <none>

&m: {}
--------------------------------------------
Context : {b1, b2 : bool}
Bound   : [=] 1%r / 2%r

pre = b1

(1)  b2 <$ {0,1}

post = b1 ^ b2
```

Here the bound is (b), and we can solve this goal with

```
rnd (pred1 false).
skip; progress.
smt(dbool1E). smt(). smt(). smt().
```

# First Example

The fourth subgoal is:

```
Type variables: <none>

&m: {}
------------------------------------------
Context : {b1, b2 : bool}
Bound   : [=] 1%r / 2%r

pre = true

(1)  b1 <$ {0,1}

post = !b1
```

Here the bound is (c), and we can solve this goal with

```
rnd (pred1 false).
skip; progress.
smt(dbool1E). smt(). smt().
```

## *First Example*

The fifth subgoal is:

```
Type variables: <none>

&m: {}
------------------------------------------
Context : {b1, b2 : bool}
Bound   : [=] 1%r / 2%r

pre = !b1

(1)  b2 <$ {0,1}

post = b1 ^ b2
```

Here the bound is (d), and we can solve this goal with

```
rnd (pred1 true).
skip; progress.
smt(dbool1E). smt(). smt(). smt().
```

# First Example

And the sixth and final subgoal is:

```
Type variables: <none>

&m: {}
------------------------------------------
forall _, true => 1%r / 2%r = 1%r / 2%r
```

The right side of this implication is EasyCrypt's simplification of

```
(a) * (b) + (c) * (d) = 1%r / 2%r
```

The first part of the sum is the probability that we get to a memory satisfying the postcondition via IC, and the second part is that we get there via the negation of IC. We are asked to prove that the sum of these two possibilities is the bound of the lemma. We can solve goal this using `trivial`.

## *Second Example*

Our second example is concerned with the modules

```
module P = {
  proc f(b : bool) : bool = {
    var b' : bool;
    b' <$ {0,1};
    return b /\ b';
  }
}.

module Q = {
  proc f(b : bool) : bool = {
    var b' : bool;
    b' <$ {0,1};
    return b /\ !b';
  }
}.
```

Note that both `P.f` and `Q.f` take a boolean parameter, `b`. And note the negation in the value returned by `Q.f`.

# *Second Example*

And this time we prove a pRHL judgement where the pre- and postconditions involve left-to-right implications on the parameters and results, respectively:

```
lemma P_Q_equiv :
  equiv [P.f ~ Q.f : b{1} => b{2} ==> res{1} => res{2}].
proof.
proc.
rnd (fun x => ! x).
auto; smt().
qed.
```

From `P_Q_equiv` we can prove the folowing lemma, beginning with the expected `move`:

```
lemma P_Q_leq (b1 b2 : bool) &m :
  (b1 => b2) =>
  Pr[P.f(b1) @ &m : res] <= Pr[Q.f(b2) @ &m : res].
proof.
move => b1_imply_b2.
```

# *Second Example*

This takes to goal

```
Type variables: <none>

b1: bool
b2: bool
&m: {}
b1_imply_b2: b1 => b2
--------------------------------------------
Pr[P.f(b1) @ &m : res] <= Pr[Q.f(b2) @ &m : res]
```

And its conclusion (an inequality of `Pr[...]` expressions for a pair of procedures) is also a form that `byequiv` can handle. Running

```
byequiv P_Q_equiv.
```

gives us the following two goals, both of which can be solved with `trivial`.

# *Second Example*

```
Type variables: <none>

b1: bool
b2: bool
&m: {}
b1_imply_b2: b1 => b2
--------------------------------------------
b1 => b2
```

and

```
Type variables: <none>

b1: bool
b2: bool
&m: {}
b1_imply_b2: b1 => b2
--------------------------------------------
forall &1 &2,
  (res{1} => res{2}) => res{1} => res{2}
```

## *Second Example*

To prove

```
lemma Q_true &m :
  Pr[Q.f(true) @ &m : res] = 1%r / 2%r.
proof.
byphoare (_ : b ==> res).
```

the default lemma chosen by `byphoare` won't suffice. Instead we must use the above one, which produces three goals, the first of which is the following (confusingly with `arg` instead of `b`)

```
Type variables: <none>

&m: {}
--------------------------------------------
pre = arg

    Q.f
    [=] 1%r / 2%r

post = res
```

## *Second Example*

Running `proc` takes us to

```
Type variables: <none>

&m: {}
--------------------------------------------
Context : {b, b' : bool}
Bound   : [=] 1%r / 2%r

pre = b

(1)  b' <$ {0,1}

post = b /\ !b'
```

which we can solve with

```
rnd (pred1 false).
auto; smt(dbool1E).
```

# *Second Example*

The second and third goals pertain to the pre- and postconditions, but EasyCrypt has already simplified them. For the precondition we must show that the argument (`true`) to `Q.f` in the lemma's statement is provable. For the postcondition, we must show an iff relationship between the event of the lemma's statement and the postcondition.

## *Second Example*

The conclusion of the following lemma is also supported by `byphoare`.

```
lemma Q_leq (b_ : bool) &m :
  Pr[Q.f(b_) @ &m : res] <= 1%r / 2%r.
proof.
byphoare => //.
```

This takes us to the goal

```
Type variables: <none>

b_: bool
&m: {}
--------------------------------------------
pre = true

    Q.f
    [<=] 1%r / 2%r

post = res
```

# *Second Example*

Running proc takes us to

```
Type variables: <none>

b_: bool
&m: {}
--------------------------------------------
Context : {b, b' : bool}
Bound   : [<=] 1%r / 2%r

pre = true

(1)  b' <$ {0,1}

post = b /\ !b'
```

From here, we can run

```
rnd (pred1 false).
```

which takes us to the goal

# *Second Example*

```
Type variables: <none>

b_: bool
&m: {}
--------------------------------------------
Context : {b, b' : bool}

pre = true /\ true


post =
  mu1 {0,1} false <= 1%r / 2%r &&
  forall (v : bool),
    v \in {0,1} => b /\ !v => pred1 false v
```

Note that this is a Hoare Logic (not pHL) judgement. Also note that the first conjunct of the postcondition is now an inequality, and also note the order of the implication in the second conjunct.

# *Second Example*

The understand the goal pertaining to the postcondition in the preceding example, we can consider this nonsensical modification of it:

```
op Z1 : bool. op Z2 : bool.

lemma Q_leq_bad (b_ : bool) &m :
  Pr[Q.f(b_) @ &m : Z1] <= 1%r / 2%r.
proof.
byphoare (_ : true ==> Z2).
```

The third subgoal produced by this is

```
Type variables: <none>

b_: bool
&m: {}
--------------------------------------------
forall _, Z1 => Z2
```

## *Second Example*

Finally, we can combine P_Q_leq and Q_leq, getting:

```
lemma P_leq (b_ : bool) &m :
  Pr[P.f(b_) @ &m : res] <= 1%r / 2%r.
proof.
rewrite (ler_trans Pr[Q.f(b_) @ &m : res]).
by rewrite P_Q_leq.
rewrite Q_leq.
qed.
```