EasyCrypt Cheat Sheet

Probability

The probability of an event `E` in a procedure `M.f` applied to a argument `a` starting from a memory `&m` is denoted by `Pr[M.f(a) @ &m : E]`.

The initial memory `&m` contains the value of global variables.

The event is an expression that can depend of global variables and a special variable `res` denoting the returned result of the procedure.

Hoare logic

The Hoare logic allows to prove properties on the result of a procedure.

A statement in this logic uses the following syntax:

```
lemma myFirstLemma : hoare[M.f : pre ==> post]
```

The pre-condition `pre` is a proposition than can depend of global variables, parameters and a special name `arg` denoting the arguments of the procedure.

The post-condition `post` can depend of global variables, and the special variable `res`.

The judgment imposes that any state with a non zero probability in the distribution generated by the procedure `M.f` starting from a memory and arguments satisfying the pre-condition will satisfy the post-condition.

For example for the program on the left, the Hoare judgements on the right are equivalent.

```
module M = {
  var v1 : int
  var v2 : int

  proc f(x:int; y: int) = {
    v1 <- 0;
    return x + y;
  }
}.
```

```
lemma version1 : hoare[M.f : x=1 /\ y=2 ==> res = 3 /\ M.v1 = 0].
lemma version2 : hoare[M.f : arg=(1,2) ==> res = 3 /\ M.v1 = 0].
```

You can also relate initial and final values using logical variables:

```
lemma relate : forall _x _y _v2, hoare[M.f : arg=(_x,_y) /\ M.v2 = _v2 ==> res=_x + _y /\ M.v2=_v2].
```

Here, the result is the sum of the arguments; global variable `M.v2` unchanged.

To make the declaration shorter you can also use the following syntax:

```
hoare relate _x _y _v2 : M.f : arg=(_x,_y) /\ v2 = _v2 ==> res=_x + _y /\ v2=_v2.
```

Probabilistic Hoare logic

Probabilistic Hoare logic allows bounding the probability of an event.

```
module P = {
  proc s() = {
    var r;
    r <$ {0,1};
    return r;
  }
}.
```

Statement: probability that the result is true equals 1/2.

```
lemma Ps_half_eq : phoare[P.s : true ==> res] = 0.5.
```

We can also express probability lower an upper bounds.

```
lemma Ps_half_le : phoare[P.s : true ==> res] <= 0.5.
lemma Ps_half_ge : phoare[P.s : true ==> res] => 0.5.
```

We can prove probability statements using pHoare judgments by applying the `byphoare` tactic.

```
lemma pr_Ps_half_eq &m : Pr[P.s() @ &m : res] = 0.5.
proof. byphoare. done. done. apply Ps_half_eq. qed.
```

A shorter version of the proof is:

```
lemma pr_Ps_half_eq &m : Pr[P.s() @ &m : res] = 0.5.
proof. byphoare Ps_half_eq => //. qed.
```

EasyCrypt provides a special notation `islossless P.f` for judgments that require only that the program always terminates.

```
islossless P.f := phoare[ P.f : true ==> true] = 1.0.
```

Probabilistic Relational Hoare logic

The main ingredient of EasyCrypt is the Probabilistic Relational Hoare logic (pRHL), which allows describing events involving the execution of two programs. The general syntax is

```
equiv[M1.f ~ M2.g : pre ⟹ post]
```

The pre-condition pre is a relation between the initial state (arguments and global variables).

The post-condition is a relation between the final states (result of each procedure and global variables).

```
module Pn = {
  proc s() = {
    var r;
    r ←$ {0,1};
    return ¬r;
  }
}
```

We can relate the executions of P.s and Pn.s:

```
lemma myFirstEquiv : equiv [P.s ~ Pn.s : true ⟹ ¬res{1} = res{2}].
```

res{1} denotes the result of the left-hand procedure (i.e. P.s).
res{2} denotes the result of the right-hand procedure (i.e. Pn.s).
Statement: P.s output is the negation of Pn.s output.

EasyCrypt provides a special notation for variable equalities: ={x, y} is a shortcut for x{1} = x{2} ∧ y{1} = y{2}.

We can prove probability statements involving two programs using pRHL judgements with the **byequiv** tactic.

PROBABILITY EQUALITIES. For example, assume we have proved **equiv** eq_M_N : M.f ∼ N.g : pre ⟹ post.

Then **byequiv** can be used to discard the following proof goal.

```
lemma pr_eq_M_N &m1 a1 &m2 a2 : Pr[M.f(a1) @ &m1 : E1] = Pr[N.g(a2) @ &m2 : E2].
proof. byequiv eq_M_N.
```

The tactic will generate two side conditions:

i. the pre-condition pre is satisfied by a1, &m1, a2, &m2; and

ii. the post-condition post implies the equivalence of the two events: post ⇒ E1 ⇔ E2.

PROBABILITY EQUALITIES. Tactic **byequiv** recognizes a number of patterns, such as inequalities:

```
lemma pr_eq_M_N &m1 a1 &m2 a2 : Pr[M.f(a1) @ &m1 : E1] ≤ Pr[N.g(a2) @ &m2 : E2].
proof. byequiv eq_M_N.

lemma pr_eq_M_N &m1 a1 &m2 a2 : Pr[M.f(a1) @ &m1 : E1] ≥ Pr[N.g(a2) @ &m2 : E2].
proof. byequiv eq_M_N.
```

In these cases the postcondition subgoals change to post ⇒ E1 ⇒ E2 and post ⇒ E2 ⇒ E1, respectively.

For clarity, here are two additional examples of using the **byequiv** tactics for probability equalities:

```
lemma rel &m1 &m2: Pr[P.s() @ &m1 : ¬res] = Pr[Pn.s() @ &m2 : res].
proof. byequiv myFirstEquiv. done. done. qed.
```

```
axiom mySecondEquiv : equiv [P.s ~ Pn.s : true ⟹ res{1} = res{2}].

lemma rel2 &m1 &m2: Pr[P.s() @ &m1 : res] = Pr[Pn.s() @ &m2 : res].
proof. byequiv mySecondEquiv ⇒ //. qed.
```

Shorter proof **byequiv** mySecondEquiv ⇒ // works in both, as ⇒ // tries to apply **done** to the subgoals.

UP-TO-BAD RELATIONS. The fundamental theorem of game-hopping is native in the EasyCrypt pRHL logic.

Rather than conditional probabilities, the following general derivation is used:

Pr[G1 : ¬bad1 ∧ E1] = Pr[G2 : ¬bad2 ∧ E2] ⟹ Pr[G1 : bad1] = Pr[G2 : bad2] ⟹ `|Pr[G1 : E1] − Pr[G2 : E2]| ≤ Pr[G2 : bad2]

Tactic **byequiv** recognizes goals such as the consequence above:

```
lemma upto &m : `|Pr[G1 : E1] − Pr[G2 : E2]| ≤ Pr[G2 : bad2].
proof. byequiv: bad1.
```

This yields one sub-goal, where the hypotheses are proved using **equiv** and pre is inferred automatically:

equiv [G1 ∼ G2 : pre ⟹ (bad1{1} ⇔ bad2{2}) ∧ (¬bad2{2} ⇒ E1{1} ⇔ E2{2})]

For pattern Pr[G1 : E1] ≤ Pr[G2 : E2] + Pr[G2 : bad2], tactic **byequiv** generates subgoal justified by the derivation below.

```
equiv [G1 ~ G2 : pre ⟹ ¬bad2{2} ⇒ E1{1} ⇒ E2{2}]]
```

Pr[G1 : E1] ≤ Pr[G2 : ¬bad2 ∧ E2 ∨ bad2] ⟹ Pr[G1 : E1] ≤ Pr[G2 : ¬bad2 ∧ E2] + Pr[G2 : bad2] ≤ Pr[G2 : E2] + Pr[G2 : bad2]

Combining one-sided and two-sided (relational) results

The HL, pHL and pRHL logics can be combined together using tactic **conseq**.

REFINING PHL WITH HL. Assume we have proved using pHL that a coarse post condition post1 holds with probability r. (Note that post1 = true implies termination with probability r and **islossless** is a particular case.)

We can refine this post condition to post2 using HL, without thinking about probabilities and termination:

```
axiom a1 : phoare[M.f : pre1 ⟹ post1] = r.
axiom a2 : hoare [M.f : pre2 ⟹ post2].
lemma l1 : phoare[M.f : pre ⟹ post] = r.
proof. conseq a1 a2.
```

This generates two natural sub-goals: pre ⇒ pre1 ∧ pre2 and post1 ∧ post2 ⇒ post.

One can also provide the pre and post-conditions required by the rule and prove the hypotheses as subgoals:

```
lemma l1 : phoare[M.f : pre ⟹ post] = r.
proof. conseq (: pre1 ⟹ post1) (: pre2 ⟹ post2).
```

One can take the current pre- and post-conditions using underscore notation (e.g., pre1=pre and post2=post):

```
lemma l1 : phoare[M.f : pre ⟹ post] = r.
proof. conseq (: _ ⟹ post1) (: pre2 ⟹ _).
```

RESTATING PHL IN A DIFFERENT PROCEDURE USING PRHL. Assume we have proved a probabilistic statement of the form

```
axiom a1 : phoare [M.f : pre1 ⟹ post1] = r.
```

Suppose also we use pRHL to relate the events in post1 to events post2 in procedure N.g, e.g.

```
axiom a2: equiv [N.g ~ M.f : pre ⟹ post1{1} ⇔ post2{2} ].
```

Then **conseq** a1 a2 yields **phoare** [N.g : pre2 ⟹ post2] = r under side condition ∀ m2, pre2{m2} ⇒ ∃ m_1, pre ∧ pre1{m_1}.

Existential quantification allows us to provide memory m_1 in which we want to use the relational hypothesis.

This is useful when proving a1 in M.f is much easier, e.g., due to control-flow/abstract data types.

More generally, the same pattern can be proved using **equiv** post condition post if post ⇒ post2 ⇔ post1.

Other useful tactics

EasyCrypt has logics for high-level reasoning about procedures and low-level reasoning about statements.

The **proc** tactic transforms a judgment on a procedure into a judgment on its statements. The **arg** variable in the precondition is replaced by the actual parameters and the **res** variable in the post-condition is replaced by the expression returned by the procedure.

The tactics that follow are useful for reasoning over statements. They exist for both one-sided (HL, pHL) and two-sided (pRHL)goals. The simplest ones are **wp**/**sp** for the weakest/strongest pre-/post-condition.

inline Tactic **inline** M.f will inline the function M.f. **inline** * will inline (recursively) all functions. If the goal is a pRHL judgment, then one can specify **inline**{1} M.f to inline M.f in the left-hand statement, whereas **inline**{2} * will inline all procedures in the right-hand statement.

call The **call** tactic applies when the last instruction(s) is/are a procedure call. For HL, this rule is standard. Assume we have

```
axiom a1 : hoare [M.f : pre1(arg) ⟹ post1(res) ]
```

and the current goal is **hoare** [S; x ← M.f(e) : pre ⟹ post(x)]. **call** a1 will transform the goal (possibly with quantification over the modified global variables) into **hoare**[S : pre ⟹ pre1(e) ∧ ∀ r, post1(r) ⇒ post(r)]

It is also possible to provide directly the pre and post-conditions to the tactic: **call** (: pref **arg** ⟹ postf **res**).

For pRHL, suppose we have

```
axiom a2 : equiv [ M.f1 ~ N.f2 : pre2(arg{1},arg{2}) ⟹ post2(res{1},res{2}) ]
```

and the current goal is

```
equiv [ S1; x1 ← M.f1(e1) ~ S2; x2 ← N.f2(e2) : pre ⟹ post(x1,x2) ]
```

Tactic call a2 will transform the goal into

```
equiv[ c1 ~ c2 : pre ⟹ pre2(e1{1},e2{2}) ∧ ∀ r1 r2, post2(r1,r2) ⇒ post(r1,r2)]
```

Again, it is possible to provide directly the pre- and post-conditions to the tactic.

There is a special case when the called procedure are abstract, e.g, adversarial procedures.

For example, support that the goal is of the form:

```
equiv [ S1; x1 ← A(O1).f(e1) ~ S2; x2 ← A(O2).f(e2) : pre ⟹ post(x1,x2) ]
```

The call tactic then implicitly quantifies for all possible code of A and rely on the fact that the same code is being run on both sides: if the code on both sides gets the same inputs, then it will return the same outputs.

The call tactic therefore requires proving as sub-goals that the adversary gets the same original inputs to the procedure, but also that it gets the same outputs from every oracle (procedure) call it makes.

For example, let us assume only one oracle procedure on each sideO1.g/O2.g and suppose inv is a relational invariant preserved by the oracles, so we have proved:

```
equiv [O1.g ~ O2.g : ={arg} ∧ inv ⟹ ={res} ∧ inv ]
```

Then we can use call to derive

```
equiv [A(O1).f ~ A(O2).f : ={arg, glob A} ∧ inv ⟹ ={res, glob A} ∧ inv].
```

where glob A represents the global memory read/written to by A.

The rule is sound as long as A is not able to break the invariant inv between calls to its oracles, so EasyCrypt ensures that the variables of inv can not be written by A (this is declared in the type of A).

Tactic call (inv) permits transforming the original goal

```
equiv [ S1; x1 ← A(O1).f(e1) ~ S2; x2 ← A(O2).f(e2) : pre ⟹ post(x1,x2) ]
```

into a number of subgoals reminiscent of a classical HL while rule:

- the standard subgoal for call on precondition ={arg, glob A} ∧ inv

- the standard subgoal for call on postcondition ={res, glob A} ∧ inv

- for each possible oracle call, a subgoal imposing invariant preservation of the form

```
equiv[O1.g ~ O2.g : ={arg} ∧ inv ⟹ ={res} ∧ inv]
```

sim The sim tactic gives automation when the post-condition can be reduced to a conjunction of equalities over variables on two-sided statements. It often discards goals fully automatically if the two programs have similar control flow.