Using EasyCrypt*'s Ambient Logic*

These slides are an example-based introduction to the use of EasyCrypt's ambient logic.

See Chapter 2 and Sections 3.1–3.3 of the EasyCrypt reference manual for more information:
https://www.easycrypt.info/documentation/refman.pdf

Types

EasyCrypt's types include basic types like `unit` (which only has the single element `()`), `int`, `bool` and `real`, as well as product types $\mathtt{t}_1 * \mathtt{t}_2 \cdots * \mathtt{t}_n$ and function types $\mathtt{t}_1$ `->` $\mathtt{t}_2$. `*` has higher precedence than `->`, and `->` is right associative.

Thus, e.g., $\mathtt{t}_1 * \mathtt{t}_2$ `->` $\mathtt{t}_3$ `->` $\mathtt{t}_4$ means $(\mathtt{t}_1 * \mathtt{t}_2)$ `->` $(\mathtt{t}_3$ `->` $\mathtt{t}_4)$. A value of this type is a function that takes in a pair `(x,y)`, where `x` has type $\mathtt{t}_1$ and `y` has type $\mathtt{t}_2$, and returns a function that takes in a value `z` of type $\mathtt{t}_3$, and returns a result of type $\mathtt{t}_4$.

Operators

EasyCrypt has typed *operators* (or functions). E.g.,

```
op f (x y : int) = x - 2 * y.
op g (a b : bool) = !(a /\ b) /\ (a \/ b).
op h : int -> bool.
```

Here the prefix operator `!` is boolean negation, and the infix operatos `/\` and `\/` are conjunction and disjunction, respectively. We also have implication `=>` and if-and-only-of `<=>`. `f` and `g` have (curried) types:

```
f : int -> int -> int
g : bool -> bool -> bool
```

Thus you can say

```
op p : int -> int = f 4.
op y : int = p 5.
```

in which case the value of `y` will be −6.

Operators

If `x` is a value of type `int`, then `x%r` is the corresponding element of `real`. Operators in EasyCrypt can be overloaded, so that, e.g., `*` is multiplication for both `int` and `real`.

Axioms and Lemmas

We can state axioms like

```
axiom h_ax (x : int) : x <> 0 => h x.
```

which says that for all non-zero integers `x`, the result of applying `h` to `x` returns `true`, i.e., `h x` holds.

We can state and prove lemmas like

```
lemma not_or (a b : bool) : !(a \/ b) => !a /\ !b.
proof.
...
qed.
```

which says that the negation of the disjunction of `a` and `b` implies the conjunction of the negation of `a` and the negation of `b`.
Here the `...` should consist of a sequence of tactics proving the lemma.

Theories

EasyCrypt has various *theories* in its standard library, each of which contains operators, axioms, lemmas and subtheories. See the subdirectory `theories` of the EasyCrypt distribution.

The theory `AllCore` contains some core theories like `Int` and `Real`—corresponding to the integers and real numbers.

Issuing the command

```
require import T.
```

makes the definitions of the theory `T` available without qualification (so you can say `f` instead of `T.f`).

Printing and Searching

Operators, lemmas and axioms may be printed using the `print` command:

```
print g.
print [!].
print (/\).
```

Note the special way unary and binary operators are specified.

The `search` command can be used to search for all lemmas and axioms involving all of a list of operators. E.g.,

```
search [!] (\/) (=>).
```

searches for all lemmas involving all of negation, disjunction and implication.

Proof Process

At each point of proving a lemma, we have some number of *goals*, and are focused on one of them. Goals consist of an ordered set of *assumptions* (listed above the horizontal bar) plus a single *conclusion* (listed below the bar).

When working on a proof, one may temporarily accept a goal, without proof, by running the tactic `admit`.

Basic Tactics

The conclusion of a goal can be logically simplified using the tactic `simplify`. E.g., `simplify` transforms

```
Type variables: <none>

x : int
y : int
------------------------------------------
!true \/ x < y
```

into

```
Type variables: <none>

x : int
y : int
------------------------------------------
x < y
```

Basic Tactics

The `trivial` tactic applies a set of basic logical rules, and can solve certain goals, e.g.:

```
Type variables: <none>

a : bool
b : bool
------------------------------------------
a => ! (true /\ b) => !true \/ !b
```

and (because it can establish a contradiction)

```
Type variables: <none>

a : bool
b : bool
not_and: ! (a /\ b)
a_true: a
b_true: b
------------------------------------------
a /\ b => false
```

SMT Solvers

The `smt` tactic uses the known SMT solvers to try to solve a goal, using all known lemmas.

Running `smt()` means to only use lemmas built-in to the solvers.

One can also list the previously proved lemmas that may be used, e.g., smt(foo goo).

One can restrict which solvers may be used, e.g.,

```
prover ["Z3" "Alt-Ergo"].
```

One can customize the timeout (in seconds) before an application of `smt` will fail:

```
timeout 2.
```

Introduction Patterns: Simple

Introduction patterns may be used to introduce into the goal's assumptions universally quantified variables as well as the left sides of implications. E.g., `move => x y z le_x_y le_y_z` transforms

```
Type variables: <none>

--------------------------------------------
forall (x y z : int), x <= y => y <= z => x <= z
```

into

```
Type variables: <none>

x : int
y : int
z : int
le_x_y: x <= y
le_y_z: y <= z
--------------------------------------------
x <= z
```

Introduction Patterns: Elimination

Introduction patterns may be used to eliminate disjunctions, existentially quantified formulas, and conjunctions on the left sides of implications. E.g., `move => []` transforms

```
Type variables: <none>

a : bool
b : bool
------------------------------------------
a \/ b => a
```

into

Introduction Patterns: Elimination

```
Type variables: <none>

a : bool
b : bool
------------------------------------------
a => a
```

and

```
Type variables: <none>

a : bool
b : bool
------------------------------------------
b => a
```

Introduction Patterns: Elimination

And `move => []` transforms

```
Type variables: <none>

y : int
---------------------------------------------
(exists (x : int), y = x * 2 + 1) =>
exists (z : int), y - 3 = z * 2
```

into

```
Type variables: <none>

y : int
---------------------------------------------
forall (x : int),
  y = x * 2 + 1 =>
  exists (z : int), y - 3 = z * 2
```

Introduction Patterns: Elimination

And `move => []` transforms

```
Type variables: <none>

a : bool
b : bool
------------------------------------------------------------------------
a /\ b => a
```

into

```
Type variables: <none>

a : bool
b : bool
------------------------------------------------------------------------
a => b => a
```

Elimination

One can do elimination of an assumption using `elim`. E.g.,

```
elim H.
```

transforms

```
Type variables: <none>

a : bool
b : bool
H : a \/ b
------------------------------------------
a
```

into

Elimination

```
Type variables: <none>

a : bool
b : bool
-------------------------------------------
a => a
```

and

```
Type variables: <none>

a : bool
b : bool
-------------------------------------------
b => a
```

Introduction Patterns Following Arbitrary Tactic

Any tactic may be followed by an introduction pattern, which applies to the subgoals created by running the tactic. And one may specify different introduction patterns for different subgoals. E.g.,

```
elim H => [a_true | b_true].
```

transforms

```
Type variables: <none>

a : bool
b : bool
H : a \/ b
--------------------------------------------
a
```

into

Introduction Patterns Following Arbitrary Tactic

```
Type variables: <none>

a : bool
b : bool
a_true: a
------------------------------------------
a
```

and

```
Type variables: <none>

a : bool
b : bool
b_true: b
------------------------------------------
a
```

Case Analysis

The case tactic can be used to do case analysis. E.g.,

```
case a
```

transforms

```
Type variables: <none>

a : bool
b : bool
------------------------------------------
! (a /\ b) => !a \/ !b
```

into

Case Analysis

```
Type variables: <none>

a : bool
b : bool
------------------------------------------
a => ! (true /\ b) => !true \/ !b
```

and

```
Type variables: <none>

a : bool
b : bool
------------------------------------------
!a => ! (false /\ b) => !false \/ !b
```

Introduction Patterns: Simplify and Trivial

Including `/=` (resp., `//`, `/#`) in an introduction pattern means apply `simplify` (resp., `trivial`, `smt()`) to the current goal. E.g.,

```
case a => //
```

solves the goal

```
Type variables: <none>

a : bool
b : bool
------------------------------------------
! (a /\ b) => !a \/ !b
```

Splitting Conjunctions

When a goal's conclusion is a conjunction, if-and-only-if or equality of tuples, it may be split into multiple subgoals using `split`. E.g., `split` transforms the goal

```
Type variables: <none>

a : bool
b : bool
not_or: ! (a \/ b)
------------------------------------------
!a /\ !b
```

into

Splitting Conjunctions

```
Type variables: <none>

a : bool
b : bool
not_or: ! (a \/ b)
------------------------------------------
!a
```

and

```
Type variables: <none>

a : bool
b : bool
not_or: ! (a \/ b)
------------------------------------------
!b
```

Splitting Conjunctions

And `split` transforms the goal

```
Type variables: <none>

a : bool
b : bool
------------------------------------------
! (a \/ b) <=> !a /\ !b
```

into

Splitting Conjunctions

```
Type variables: <none>

a : bool
b : bool
------------------------------------------
! (a \/ b) => !a /\ !b
```

and

```
Type variables: <none>

a : bool
b : bool
------------------------------------------
!a /\ !b => ! (a \/ b)
```

Splitting Conjunctions

And `split` transforms the goal

```
Type variables: <none>

x : int
x': int
y : bool
y': bool
eq_x_x': x = x'
eq_y_y': y = y'
------------------------------------------
(x, y) = (x', y')
```

into

Splitting Conjunctions

```
Type variables: <none>

x : int
x': int
y : bool
y': bool
eq_x_x': x = x'
eq_y_y': y = y'
------------------------------------------
x = x'
```

and

```
Type variables: <none>

x : int
x': int
y : bool
y': bool
eq_x_x': x = x'
eq_y_y': y = y'
------------------------------------------
y = y'
```

Proving Disjunctions

The tactics `left` and `right` can be used to prove disjunctions. E.g., `left` transforms the goal

```
Type variables: <none>

a : bool
b : bool
a_true: a
------------------------------------------
a \/ b
```

into

```
Type variables: <none>

a : bool
b : bool
a_true: a
------------------------------------------
a
```

Proving Disjunctions

And `right` transforms the goal

```
Type variables: <none>

a : bool
b : bool
b_true: b
------------------------------------------
a \/ b
```

into

```
Type variables: <none>

a : bool
b : bool
b_true: b
------------------------------------------
b
```

Proving Existentially Quantified Formulas

The tactic `exists` can be used to prove existentially quantified formulas. E.g., `exists (x - 1)` transforms the goal

```
Type variables: <none>

y : int
x : int
y_eq: y = x * 2 + 1
------------------------------------------
exists (z : int), y - 3 = z * 2
```

into

```
Type variables: <none>

y : int
x : int
y_eq: y = x * 2 + 1
------------------------------------------
y - 3 = (x - 1) * 2
```

Proving Sublemmas

When working on proving a goal, one may prove and then use a sublemma using the tactic `have`. E.g.,

```
have : a \/ b.
```

transforms the goal

```
Type variables: <none>

a : bool
b : bool
not_or: ! (a \/ b)
a_true: a
------------------------------------------
false
```

into the two subgoals

Proving Sublemmas

```
Type variables: <none>

a : bool
b : bool
not_or: ! (a \/ b)
a_true: a
------------------------------------------
a \/ b
```

and

```
Type variables: <none>

a : bool
b : bool
not_or: ! (a \/ b)
a_true: a
------------------------------------------
a \/ b => false
```

Proving Sublemmas

What comes before `have` is an arbitrary introduction pattern to be applied to the second subgoal. E.g.,

```
have contrad : a \/ b.
```

transforms the goal

```
Type variables: <none>

a : bool
b : bool
not_or: ! (a \/ b)
a_true: a
------------------------------------------
false
```

into the two subgoals

Proving Sublemmas

```
Type variables: <none>

a : bool
b : bool
not_or: ! (a \/ b)
a_true: a
------------------------------------------
a \/ b
```

and

```
Type variables: <none>

a : bool
b : bool
not_or: ! (a \/ b)
a_true: a
contrad: a \/ b
------------------------------------------
false
```

Applying Lemmas

We can apply an already proven lemma using the `apply` tactic; it can also be used to apply an assumption. E.g., if we've already proved

```
lemma not_or_imp (a b : bool) : !(a \/ b) => !a /\ !b.
```

then running

```
apply (not_or_imp (x < y) (y < x)).
```

solves the goal

```
Type variables: <none>

x : int
y : int
----------------------------------------------
! (x < y \/ y < x) => ! x < y /\ ! y < x
```

Applying Lemmas

And EasyCrypt can often infer the instantiations of the applied lemma's parameters. E.g., running

```
apply not_or_imp.
```

solves the goal

```
Type variables: <none>

x : int
y : int
-------------------------------------------
! (x < y \/ y < x) => ! x < y /\ ! y < x
```

Applying Lemmas

Furthermore, if we've proved an if-and-only-iff lemma, we can apply it in place of either the left-to-right or right-to-left implications. E.g., if we have

```
lemma not_or_iff (a b : bool) : !(a \/ b) <=> !a /\ !b.
```

then running

```
apply not_or_iff.
```

solves the goal

```
Type variables: <none>

x : int
y : int
---------------------------------------------
! (x < y \/ y < x) => ! x < y /\ ! y < x
```

Rewriting Equational Lemmas

If we have equational lemmas like

```
lemma foo (x : int) : f x = x + 1
```

where the operator `f` has type `int -> int`, we can rewrite them in formulas using the `rewrite` tactic.

E.g., the tactic

```
rewrite (f_eq x).
```

transforms the goal

```
Type variables: <none>

x : int
y : int
-------------------------------------------
f (f x * f y) = (x + 1) * (y + 1) + 1
```

into

Rewriting Equational Lemmas

```
Type variables: <none>

x : int
y : int
------------------------------------------
f ((x + 1) * f y) = (x + 1) * (y + 1) + 1
```

Rewriting Equational Lemmas

And the tactic

```
rewrite (f_eq y).
```

transforms the goal

```
Type variables: <none>

x : int
y : int
--------------------------------------------
f ((x + 1) * f y) = (x + 1) * (y + 1) + 1
```

into

```
Type variables: <none>

x : int
y : int
--------------------------------------------
f ((x + 1) * (y + 1)) = (x + 1) * (y + 1) + 1
```

Rewriting Equational Lemmas

And the tactic

```
f_eq ((x + 1) * (y + 1)).
```

transforms the goal

```
Type variables: <none>

x : int
y : int
--------------------------------------------
f ((x + 1) * (y + 1)) = (x + 1) * (y + 1) + 1
```

into

```
Type variables: <none>

x : int
y : int
--------------------------------------------
(x + 1) * (y + 1) + 1 = (x + 1) * (y + 1) + 1
```

Rewriting Equational Lemmas

As with `apply`, `rewrite` can often infer the parameters of the equational lemma, and we can also do rewriting from right-to-left by prepending a `-`. E.g., the tactic

```
rewrite -f_eq -f_eq -f_eq
```

transforms the goal

```
Type variables: <none>

x : int
y : int
------------------------------------------------------------------------
f (f x * f y) = (x + 1) * (y + 1) + 1
```

into

```
Type variables: <none>

x : int
y : int
------------------------------------------------------------------------
f (f x * f y) = f (f x * f y)
```

Rewriting Equational Lemmas

The `rewrite` tactic can also be used with conditional equational lemmas like

```
lemma f_eq (x : int) :
  0 <= x => f x = x + 1
```

In this case,

```
rewrite f_eq.
```

transforms the goal

```
Type variables: <none>

x : int
y : int
ge0_x: 0 <= x
ge0_y: 0 <= y
------------------------------------------
f (x + y) = x + y + 1
```

into

Rewriting Equational Lemmas

```
Type variables: <none>

x : int
y : int
ge0_x: 0 <= x
ge0_y: 0 <= y
-------------------------------------------
0 <= x + y
```

and

```
Type variables: <none>

x : int
y : int
ge0_x: 0 <= x
ge0_y: 0 <= y
-------------------------------------------
x + y + 1 = x + y + 1
```

Rewriting Equational Lemmas

If we combine conditional equational lemmas $\mathtt{l}_1$ and $\mathtt{l}_2$ in a single use of `rewrite`, then $\mathtt{l}_2$ is rewritten in the conclusion of every subgoal generated by the rewriting of $\mathtt{l}_1$ in the conclusion of the original goal.

We can say, e.g.,

```
rewrite {3}l.
```

to rewrite `l` in the current goal's conclusion in only the third applicable position.

We can also say

```
rewrite l in H.
```

to rewrite `l` is the assumption `H`.

Rewriting Equational Lemmas

We can include `//`, `/=` and `/#` in rewriting, to apply `trivial`, `simplify` and `smt()`, respectively.

If an operator `f` has a concrete definition, e.g.,

```
op f(x : int) = x * 2 - 1.
```

Then `rewrite /f` substitutes `f`'s argument for its parameter(s) in its body (`x * 2 - 1` in this case).

Rewriting Nonequational Lemmas

Rewrite can also be used with non-equational lemmas, rewriting the conclusion of the lemma (what we get after introducing all universally quantified variables and left sides of implications) with `true`. E.g., if we have

```
axiom f_ax (x : int) : 3 <= x => f x.
```

then `rewrite f_ax` transforms

```
Type variables: <none>

x : int
y : int
lt_x_y: x < y
le_3_x: 3 <= x
------------------------------------------
f x /\ x < y + 1
```

into

Rewriting Nonequational Lemmas

```
Type variables: <none>

x : int
y : int
lt_x_y: x < y
le_3_x: 3 <= x
------------------------------------------
3 <= x
```

and

```
Type variables: <none>

x : int
y : int
lt_x_y: x < y
le_3_x: 3 <= x
------------------------------------------
true /\ x < y + 1
```

Progress

The `progress` tactic uses other tactics like application of introduction patterns and `split` to reduce the current goal to one of more subgoals. E.g., `progress` reduces

```
Type variables: <none>

k': int
n': int
n : int
r : int
k : int
--------------------------------------------
((0 < k' /\ n' ^ k' * r = n ^ k) /\ 1 < k') /\
f k' =>
0 < g k' 2 /\ (n' * n') ^ g k' 2 * r = n ^ k
```

to the two subgoals

Progress

```
Type variables: <none>

k': int
n': int
n : int
r : int
k : int
H : 0 < k'
H0: n' ^ k' * r = n ^ k
H1: 1 < k'
H2: f k'
------------------------------------------
0 < g k' 2
```

and

Progress

```
Type variables: <none>

k': int
n': int
n : int
r : int
k : int
H : 0 < k'
H0: n' ^ k' * r = n ^ k
H1: 1 < k'
H2: f k'
------------------------------------------
(n' * n') ^ g k' 2 * r = n ^ k
```

Sequencing Tactics

If $\mathtt{t}_1$ and $\mathtt{t}_2$ are tactics, then $\mathtt{t}_1\mathtt{;}\mathtt{t}_2$ applies $\mathtt{t}_2$ to all the subgoals (if any) generated by running $\mathtt{t}_1$. Sequencing groups to the left, so that $\mathtt{t}_1\mathtt{;}\mathtt{t}_2\mathtt{;}\mathtt{t}_3$ means $(\mathtt{t}_1\mathtt{;}\mathtt{t}_2)\mathtt{;}\mathtt{t}_3$.

E.g., `t;trivial` applies `trivial` to every subgoal generated by running `t`.