Using EasyCrypt*'s Ambient Logic*

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

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) = if 0 < x then x - 2 * y else 1.
op g (a b : bool) = !(a /\ b) /\ (a \/ b).
op h : int -> bool.
```

Note how we can use conditionals in expressions. 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.

We can select a component of a tuple with the .` notation. E.g., (1,3,5).`2 is equal to 3.

EasyCrypt has anonymous functions. Then, if we write

```
op f : int -> bool = fun (x : int) => x = 0.
op h : (int -> bool) -> bool = fun (f : int -> bool) => f 3.
op x : bool = h f.
```

we have that f is the function that takes in an argument f of type int -> bool, and applies f to 3, returning the boolean result.

Consequently, we have that x evalutes to false.

Operators

EasyCrypt also has `let` expressions. E.g., you can write

```
op x : int = let y = 10 in y * y.
```

This binds `y` to 10 in the expression `y * y`, so that the value of `x` will then be 100.

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`). Leaving out `import` makes them available, but with qualification.

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. If an operator is an abbreviation (printing it will tell you this), you'll have to search for what it's defined to be.

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).

EasyCrypt provides various *tactics*, which reduce a goal to zero or more subgoals. When we apply a tactic to the current goal, the generated subgoals will have to be proved before the other preexisting goals are proved.

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
```

Basic Tactics

`simplify` and `trivial` never fail, although they may leave a goal unchanged, i.e., they may fail to make any progress.

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 quorum=2 ["Z3" "Alt-Ergo"].
```

says that both Z3 and Alt-Ergo must agree on each use of `smt`. Removing `quorum=2` means `smt` will succeed if either or both of the provers solve the goal.

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: Simple

If an assumption won't be needed, one can use _ instead of an identifier. And already introduced assumptions can be removed using `clear` (e.g., `clear le_y_x.`).

E.g., `move => x y z le_x_y _` transforms

```
Type variables: <none>

----------------------------------------
forall (x y z : int),
  x <= y => y <= z => x + 1 <= y + 1
```

into

```
Type variables: <none>

x: int
y: int
z: int
le_x_y: x <= y
----------------------------------------
x + 1 <= y + 1
```

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
```

(The latter goal won't be provable.)

Introduction Patterns: Elimination

And we may give different introduction patterns for the disjuncts.
E.g., `move => [a_true | b_true]` transforms

```
Type variables: <none>

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

into

Introduction Patterns: Elimination

```
Type variables: <none>

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

and

```
Type variables: <none>

a: bool
b: bool
b_true: 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 => [x y_eq]` 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
x: int
y_eq: 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
```

Introduction Patterns: Elimination

And `move => [a_true b_true]` transforms

```
Type variables: <none>

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

into

```
Type variables: <none>

a: bool
b: bool
a_true: a
b_true: 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 all the goals generated by applying the preceding parts of the introduction pattern to the tactic at hand. E.g.,

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

solves the goal

```
Type variables: <none>

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

The `by` *Tactic*

If `t` is a tactic, then

```
by t
```

means apply `trivial` to all of the goals (if any) generated by `t`, and succeed if and only all of those goals are solved by `trivial`.

E.g.,

```
by 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
```

Proving Sublemmas

When the proof of a sublemma has the form `by t`, where `t` is a tactic, the dot at the end of the `have` can be omitted. E.g.,

```
have lt_1_3 : 1 < 3.
  by trivial.
...
```

can be contracted to

```
have lt_1_3 : 1 < 3 by trivial.
...
```

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
```

Parameters that EasyCrypt should be able to infer can be written as _, and only instantiating some of the parameters may sometimes suffice.

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
```

Applying Lemmas

We can also apply a lemma to a goal when the conclusion of the lemma matches the goal's conclusion. E.g., if we have

```
lemma goo (x : int) :
  0 <= x => x <= 10 => 0 <= 2 * x /\ 2 * x <= 20.
```

then

```
apply goo
```

reduces the goal

```
Type variables: <none>

x: int
y: int
H1: 2 <= x + y + 2
H2: x + y + 2 <= 12
-------------------------------------------
0 <= 2 * (x + y) /\ 2 * (x + y) <= 20
```

to the goals

Applying Lemmas

```
Type variables: <none>

x: int
y: int
H1: 2 <= x + y + 2
H2: x + y + 2 <= 12
-------------------------------------------
0 <= x + y
```

and

```
Type variables: <none>

x: int
y: int
H1: 2 <= x + y + 2
H2: x + y + 2 <= 12
-------------------------------------------
x + y <= 10
```

Rewriting Equational Lemmas

If we have equational lemmas like

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

where the operator `f` has type `int -> int`, we can rewrite them in formulas using the `rewrite` tactic. We can also use `rewrite` with assumptions that are equations.

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. We can also do rewriting from right-to-left by prepending a `-`. And we can combine multiple rewritings into a single application of `rewrite`.

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

Rewriting Equational Lemmas

```
Type variables: <none>

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

Rewriting Equational Lemmas

We can also use `rewrite` to rewrite an if-and-only-if lemma or assumption either forward or backward, treating it like an equation.

E.g., suppose we have proved the lemma

```
lemma foo (x y : int) :
  x < y <=> x + 1 < y + 1.
```

Then

```
rewrite foo.
```

transforms the goal

```
Type variables: <none>

x: int
y: int
H: x + 1 + 1 < y + 1
-------------------------------------------
x + 1 < y \/ y < x
```

into

Rewriting Equational Lemmas

```
Type variables: <none>

x: int
y: int
H: x + 1 + 1 < y + 1
--------------------------------------------
x + 1 + 1 < y + 1 \/ y < x
```

and then

```
rewrite -foo.
```

transforms that goal back into

Rewriting Equational Lemmas

```
Type variables: <none>

x: int
y: int
H: x + 1 + 1 < y + 1
-------------------------------------------
x + 1 < y \/ y < x
```

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

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 say, e.g.,

```
rewrite 2!l.
```

to rewrite `l` twice. This will only be necessary when the second opportunity for using `l` is exposed by the first one, or different type variable instantiation is involved.

Rewriting Equational Lemmas

We can also say

```
rewrite l in H.
```

to rewrite `l` is the assumption `H`. If `l` is also an assumption, this will only be allowed if `l` appears before `H` is the list of assumptions.

Rewriting Equational Lemmas

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). If `f` isn't applied to arguments, it will be replaced by the anonymous function corresponding to its defintion. E.g., running

```
rewrite /f
```

reduces the goal

```
Type variables: <none>

x: int
x_eq: x = 10
--------------------------------------------
f (x + 1) = 21
```

to the goal

Rewriting Equational Lemmas

```
Type variables: <none>

x: int
x_eq: x = 10
--------------------------------------------
(x + 1) * 2 - 1 = 21
```

which is solved by running

```
by rewrite x_eq.
```

Rewriting Nonequational Lemmas

Forward rewriting 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) to `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
```

If rewriting results in the conclusion `true`, then the goal is solved.

Rewriting Nonequational Lemmas

If, instead, the conclusion of the lemma is a negation, then the negated formula is replaced by `false` in the goal's conclusion. 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
--------------------------------------------
false \/ x < y + 1
```

Combining Conditional Rewritings

If we combine conditional lemmas `l1` and `l2` in a single use of `rewrite`, then `l2` is rewritten in the conclusion of every subgoal generated by the rewriting of `l1` in the conclusion of the original goal.

If we only want to apply `l2` to, say, the first subgoal generated by `l1`, we can use

```
rewrite l1 1:l2.
```

This generalizes to a sequence of more then two rewritings, with subsequent rewritings being applied to all unsolved goals of the previous steps.

We can include `//`, `/=` and `/#` in rewriting, to apply `trivial`, `simplify` and `smt()`, respectively, to all the goals generated by previous rewriting steps.

Combining Conditional Rewritings

For example, running

```
rewrite H3.
```

reduces the goal

```
Type variables: <none>

a: bool
b: bool
c: bool
d: bool
H1: b
H2: a
H3: c => a => d
H4: b => c
------------------------------------------
d
```

to the goals

Combining Conditional Rewritings

```
Type variables: <none>

a: bool
b: bool
c: bool
d: bool
H1: b
H2: a
H3: c => a => d
H4: b => c
-------------------------------------------
c
```

and

Combining Conditional Rewritings

```
Type variables: <none>

a: bool
b: bool
c: bool
d: bool
H1: b
H2: a
H3: c => a => d
H4: b => c
---------------------------------------------
a
```

Combining Conditional Rewritings

Because `rewrite H4` is only applicable to the first of these subgoals, the following rewriting won't work:

```
rewrite H3 H4.
```

On the other hand

```
rewrite H3 1:H4.
```

works, as it only applies `rewrite H4` to the first subgoal generated by `rewrite H3`.

Rewriting Via an Introduction Pattern

We can use `->` and `<-` in an introduction pattern when a goal's conclusion is an implication whose left-hand-side is an equation, to be rewritten in the right-hand-side of the implication. The equation is rewritten in the forward direction with `->`, and in the backward direction with `<-`.

E.g., `move => ->` transforms

```
Type variables: <none>

x: int
y: int
-------------------------------------------
x = y + 1 => x + 1 = y + 2
```

into

Rewriting Via an Introduction Pattern

```
Type variables: <none>

x: int
y: int
--------------------------------------------
y + 1 + 1 = y + 2
```

Using, e.g., {2}-> or {3}<- allows us to say which occurrences we want to rewrite.

Rewriting Via an Introduction Pattern

And we can also use `->` for non-equational rewriting, rewriting the left-hand-side of an implication to `true`, or—in the case when the left-hand-side is a negated formula—rewriting the formula to `false`.

Rewriting Via an Introduction Pattern

E.g.,

```
move => ->.
```

transforms

```
Type variables: <none>

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

into

```
Type variables: <none>

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

Rewriting Via an Introduction Pattern

And

```
move => ->.
```

transforms

```
Type variables: <none>

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

into

```
Type variables: <none>

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

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
```

It sometimes happens that one or more of the subgoals generated by `progress` is not solvable, even though the original goal was solvable by another approach. Furthermore, if you want `progress` to treat concrete operators as opague (i.e., not to replace them by their definitions), you can run `progress [-delta]`.

Crush

An alternative to `progress` is to use the introduction pattern `/>`, which is pronouced "crush" (there is also a version that treats concrete operators as opaque: |>). Instead of generating multiple goals, it always give us a single one, which is solvable if-and-only-iff the original goal was.

E.g., `move => />` 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 goal

Crush

```
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
```

Eliminating Multiple Conjunctions

There is also an introduction pattern [#] which simply eliminates multiple conjunctions in the left side of the implication being proved. E.g., `move => [#]` 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 goal

Eliminating Multiple Conjunctions

```
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
```

You can also use [#] when the assumption is the equality between a pair of tuples.

Sequencing Tactics

If $\texttt{t}_1$ and $\texttt{t}_2$ are tactics, then $\texttt{t}_1\texttt{;}\texttt{t}_2$ applies $\texttt{t}_2$ to all the subgoals (if any) generated by running $\texttt{t}_1$. This will fail if running $\texttt{t}_2$ on even one of those subgoals fails. Sequencing groups to the left, so that $\texttt{t}_1\texttt{;}\texttt{t}_2\texttt{;}\texttt{t}_3$ means $\texttt{(t}_1\texttt{;}\texttt{t}_2\texttt{);}\texttt{t}_3$.

E.g., `t;trivial` applies `trivial` to every subgoal generated by running `t`. Because `trivial` never fails, this is always safe.

If we need to run different tactics on each of the subgoals generated by $\texttt{t}_1$, this is also possible. E.g., suppose it generates three subgoals, and we want to run $\texttt{t}_{2,1}$ on the first subgoal, $\texttt{t}_{2,2}$ on the second subgoal, and $\texttt{t}_{2,3}$ on the third subgoal, we can write $\texttt{t}_1\texttt{; [t}_{2,1} \mid \texttt{t}_{2,2} \mid \texttt{t}_{2,3}\texttt{]}$.

And `idtac` is the identity tactic, which does nothing—which is useful when we don't want to apply any tactic to one of the subgoals.

Sequencing Tactics

For example, running

```
apply H; [apply a_true | apply b_true].
```

solves the goal

```
Type variables: <none>

a: bool
b: bool
c: bool
H: a => b => c
a_true: a
b_true: b
--------------------------------------------
c
```

Case Analysis on Structured Data

The case tactic can also be used to do case analysis on structured data, like tuples. E.g., running

```
case x.
```

transforms the goal

```
Type variables: <none>

x: int * int * int
--------------------------------------------
f x = 0 => x.`1 = 0 \/ x.`2 = 0 \/ x.`3 = 0
```

to the goal

```
Type variables: <none>

--------------------------------------------
forall (x1 x2 x3 : int),
  f (x1, x2, x3) = 0 =>
  (x1, x2, x3).`1 = 0 \/
  (x1, x2, x3).`2 = 0 \/ (x1, x2, x3).`3 = 0
```

Case Analysis on Structured Data

from which running

```
move => x1 x2 x3.
```

gives us the goal

```
Type variables: <none>

x1: int
x2: int
x3: int
------------------------------------------
f (x1, x2, x3) = 0 =>
(x1, x2, x3).`1 = 0 \/
(x1, x2, x3).`2 = 0 \/ (x1, x2, x3).`3 = 0
```

Moving Assumptions back to the Goal's Conclusion

Sometimes it's useful to move assumptions back to the goal's conclusion. E.g., if we are trying to prove

```
Type variables: <none>

x: int
y: int
z: int
ge0_y: 0 <= y
ge0_z: 0 <= z
--------------------------------------------
x ^ (y + z) = x ^ y * x ^ z
```

We can run

```
move : y ge0_y
```

to get the goal

Moving Assumptions back to the Goal's Conclusion

```
Type variables: <none>

x: int
z: int
ge0_z: 0 <= z
--------------------------------------------
forall (y : int),
  0 <= y => x ^ (y + z) = x ^ y * x ^ z
```

This is now in the right form to prove by mathematical induction, given that z need not be varied in the induction.

Using Induction Principles

Various EasyCrypt theories provide induction principles. E.g., `Int` gives the principle of mathematical induction in the form of the lemma:

```
lemma intind (p : int -> bool) :
  p 0 =>
  (forall (i : int), 0 <= i => p i => p (i + 1)) =>
  forall (i : int), 0 <= i => p i.
```

To apply `intind`, our goal's conclusion must have the form

```
forall (i : int), 0 <= i => p i.
```

for some instantiation of `p` and `i`. Thus we may first need to massage our actual goal into this form.

Using Induction Principles

For example, if we run the tactic

```
elim /intind.
```

this will reduce the goal

```
Type variables: <none>

x: int
z: int
ge0_z: 0 <= z
--------------------------------------------
forall (y : int),
  0 <= y => x ^ (y + z) = x ^ y * x ^ z
```

into the goals

Using Induction Principles

```
Type variables: <none>

x: int
z: int
ge0_z: 0 <= z
--------------------------------------------
x ^ (0 + z) = x ^ 0 * x ^ z
```

(the basis step) and

Using Induction Principles

```
Type variables: <none>

x: int
z: int
ge0_z: 0 <= z
--------------------------------------------
forall (i : int),
  0 <= i =>
  x ^ (i + z) = x ^ i * x ^ z =>
  x ^ (i + 1 + z) = x ^ (i + 1) * x ^ z
```

(the inductive step). When proving the inductive step, we run

```
move => i ge0_i IH.
```

to get the goal

Using Induction Principles

```
Type variables: <none>

x: int
z: int
ge0_z: 0 <= z
i: int
ge0_i: 0 <= i
IH: x ^ (i + z) = x ^ i * x ^ z
--------------------------------------------
x ^ (i + 1 + z) = x ^ (i + 1) * x ^ z
```

Here `IH` is the inductive hypothesis.

Abstract Types

EasyCrypt lets us define abstract types and operators over those types, and to state axioms involving those operators and types. E.g., we can say

```
type t.
op f : t -> t.
axiom ax (x : t) :
  f x = f (f x).
```

Lemmas that we then prove will be valid for any instantiation of the types and operators that satisfy the axioms.

Concrete Datatypes

We can also define concrete datatypes, by listing their constructors. E.g., we can define a datatype of binary trees whose leaves are labeled by values of type `'a` and internal nodes are labeled by values of type `'b`:

```
type ('a, 'b) tree = [
| Leaf of 'a
| Node of 'b * ('a, 'b) tree * ('a, 'b) tree ].
```

Then

```
op x = Node 3 (Leaf false) (Node 2 (Leaf true) (Leaf false)).
```

is a `(bool, int) tree` whose root node is labeled by 3, with a left child consisting of a leaf labeled by `false`, and where the right child is a tree whose root node is labeled by 2, and with left and right children consisting of leaves labeled by `true` and `false`, respectively.

Concrete Datatypes

We can then recursively define the size of a tree by

```
op size (tr : ('a, 'b) tree) : int =
  with tr = Leaf x          => 1
  with tr = Node y tr1 tr2 => size tr1 + size tr2.
```

Because `x` and `y` are not used, they could be replaced by the wildcard _.

EasyCrypt gives us a structural induction principle for our datatype for free. E.g., given the goal

```
Type variables: 'a, 'b

--------------------------------------------
forall (tr : ('a, 'b) tree), 0 <= size tr
```

running

```
elim.
```

gives us the goals

Concrete Datatypes

```
Type variables: 'a, 'b

============================================
forall (x : 'a), 0 <= size (Leaf x)
```

(which can be solved using `trivial`) and

Concrete Datatypes

```
Type variables: 'a, 'b

============================================
forall (x : 'b) (t t0 : ('a, 'b) tree),
  0 <= size t =>
  0 <= size t0 => 0 <= size (Node x t t0)
```

The proof of this second goal can begin with running

```
move => x tr1 tr2 IH_tr1 IH_tr2.
```

which gives us the goal

Concrete Datatypes

```
Type variables: 'a, 'b

x: 'b
tr1: ('a, 'b) tree
tr2: ('a, 'b) tree
IH_tr1: 0 <= size tr1
IH_tr2: 0 <= size tr2
--------------------------------------------------------------
0 <= size (Node x tr1 tr2)
```

The next step should be

```
simplify.
```

which will give us the goal

Concrete Datatypes

```
Type variables: 'a, 'b

x: 'b
tr1: ('a, 'b) tree
tr2: ('a, 'b) tree
IH_tr1: 0 <= size tr1
IH_tr2: 0 <= size tr2
--------------------------------------------------------------------------
0 <= size tr1 + size tr2
```

The goal could then be solved, e.g., by running

```
smt().
```

Concrete Datatypes

Given goal

```
Type variables: 'a, 'b

tr: ('a, 'b) tree
---------------------------------------------
0 <= size tr
```

we don't need to first run

```
move : x.
```

Instead, we can directly run

```
elim tr.
```

Combining Multiple Inequalities and Using `&&` and `||`

We can chain together multiple occurrences of `<` and `<=`, as in

```
x < y <= z < w
```

which is logically equivalent to

```
x < y /\ y <= z /\ z < w
```

Actually, it's an abbreviation for

```
x < y && y <= z && z < w
```

These alternative conjunctions are equivalent to the usual ones (and we also have the alternative disjuction `||`), but EasyCrypt's tactics treat them slightly differently.

When proving `a && b` using `split`, we get a goal for proving `a` as usual. But the goal for proving `b` gives us `a` as an assumption to help us.

Combining Multiple Inequalities and Using && and ||

For example, running

```
move => [lt_x_y lt_y_z].
```

transforms the goal

```
Type variables: <none>

x: int
y: int
z: int
------------------------------------------------------------------------
x < y && y < z => x + 1 < y + 1 && y + 1 <= z
```

into

Combining Multiple Inequalities and Using `&&` and `||`

```
Type variables: <none>

x: int
y: int
z: int
lt_x_y: x < y
lt_y_z: y < z
-------------------------------------------------------------------
x + 1 < y + 1 && y + 1 <= z
```

from which running

```
split.
```

gives us the goals

```
Type variables: <none>

x: int
y: int
z: int
lt_x_y: x < y
lt_y_z: y < z
------------------------------------------------------------
x + 1 < y + 1
```

and

Combining Multiple Inequalities and Using `&&` and `||`

```
Type variables: <none>

x: int
y: int
z: int
lt_x_y: x < y
lt_y_z: y < z
------------------------------------------------------------
x + 1 < y + 1 => y + 1 <= z
```

(in this case the left-hand-side of the implication doesn't help us prove the right-hand-side).

The following lemmas let us go back and forth between the alternative conjunction and disjunction and the standard ones:

```
lemma oraE : forall (a b : bool), a || b <=> a \/ b.
lemma andaE : forall (a b : bool), a && b <=> a /\ b.
```