A Commentary for
"The Joy of EasyCrypt"

A Beginner's Guide to Formal Verification of Cryptography

v 1.0

Ji, Yong-hyeon

(`hacker3740@kookmin.ac.kr`)

Department of Information Security, Cryptology, and Mathematics

College of Science and Technology

Kookmin University

February 14, 2025

Copyright

Changelog

v1.0	2025-01-20	Initial release:

Contents

1 Computer-Aided Cryptography

2 Specification for EasyCrypt

2.1 Type Expressions

EasyCrypt 's **type expressions** are derived from four fundamental constructs:

1. type variables,
2. type constructors (or named types),
3. function types, and
4. tuple (product) types.

Type constructors may represent either built-in types or user-defined types, including record types and datatypes (variant types). Figure 0.0 outlines the syntax of these type expressions. Notably, all EasyCrypt types must be inhabited—that is, none can be empty.

τ, σ	::=	`tyvar`	type variable
	|	`_`	anonymous type variable
	|	(τ)	parenthesized type
	|	$\tau \rightarrow \sigma$	function type
	|	$(\tau_1 * \cdots * \tau_n)$	tuple type
	|	`tyname`	named type
	|	τ `tyname`	applied type constructor
	|	$(\tau_1, \cdots, \tau_n)$ `tyname`	ibid.

Figure 1: **EasyCrypt** 's type expressions

1. *A type variable.*

   These act like free variables for which a concrete type might later be substituted.

2. *An anonymous type variable.*

   Serves as a placeholder when the specific identity of a type variable is unimportant or omitted.

3. *A parenthesized type*, used purely for grouping.

   E.g. (τ) is the same as τ, but parentheses may clarify precedence.

4. *A function type.*

   In standard category-theoretic terms, this is the type $\tau \rightarrow \sigma$ of morphisms from τ to σ. It is also akin to the set of total functions from the set (or type) τ to the set (type) σ.

5. *A tuple type.*

   Interpreted as a *product type*, i.e., $\tau_1 \times \cdots \times \tau_n$.

6. *A named type.*

   This might be a base type (like `int`, `bool`, or `real`) or another type name declared in a theory. Algebraically, one can think of these as *atomic* or *defined* types.

7. *An applied type constructor* in unary form.

   For instance, if `tyname` is a unary type constructor T, then τ `tyname` is the *instantiation* of T at the type τ.

8. *An applied type constructor* taking n parameters.

   If `tyname` is an n-ary constructor (like a generic container type $\text{Map}(\cdot,\cdot)$ expecting two type parameters), then $(\tau_1, \ldots, \tau_n)$ `tyname` is its instantiation at types $\tau_1, \ldots, \tau_n$.

   Mathematically, think of this as $T(\tau_1, \ldots, \tau_n)$, where T is an n-ary functor on types.

3 EasyCrypt's Ambient Logic I

The ambient logic in **EasyCrypt** serves as the foundation for all proof scripts. To gain a deeper understanding of its functionality, we will explore several examples.

As demonstrated in the earlier motivating example, formal proofs are constructed as a sequence of proof tactics. Up to this point, we have only encountered the admit tactic. In this chapter, we will expand on this by introducing additional basic tactics and applying them to simple mathematical properties of integers.

3.1 Basic Tactics and Theorem Proving

EasyCrypt features a typed expression language, meaning every declared entity must either have an explicitly defined type or a type that can be inferred from the context. As previously noted, **EasyCrypt** provides basic built-in data types, which can be accessed by importing the relevant theories into the current environment. For this particular file, we will work with the '`Int`' theory file. Additionally, to enable **EasyCrypt** to display all the proof goals during our work, we use a directive known as a pragma.

```
require import Int.
pragma Goals: printall.
```

Code 1: Imports and pragma

Typically, the initial steps in **EasyCrypt** scripts involve importing the required theories and setting the appropriate pragmas. Before delving into cryptography, it is essential to understand how to guide **EasyCrypt** in modifying proof goals and making progress. This process is facilitated by the use of tactics. In general, the proofs for lemmas in **EasyCrypt** follow a structured approach, as illustrated below:

```
lemma name (. . .) : (. . .) .
proof.
  tactic-1.
  ...
  tactic-n.
qed.
```

Code 2: Proof script form

3.1.1 Tactic: `trivial`

We begin by exploring some basic properties of integers and demonstrating how a few key tactics operate in **EasyCrypt**.

Reflexivity, the property that any integer is equal to itself, can be expressed mathematically as:

$$\forall n \in \mathbb{Z},\ n = n.$$

In EasyCrypt, this property can be stated as a lemma and proved as follows:

```
lemma int_refl: forall (n: int), n = n.
proof.
  trivial.
qed.
```

Code 3: Using the `trivial` tactic

Once the lemma is declared and evaluated, **EasyCrypt** populates the goal pane with the statement that needs to be proved. For this lemma, the goal pane displays the reflexivity property.

```
Current goal
Type variables: <none>
----------------------------
forall (n : int), n = n
```

The proof script begins with the `proof` keyword, after which **EasyCrypt** expects the application of tactics to close the goal. In this case, we use the trivial tactic to prove the lemma `int_refl`. Upon applying `trivial`, the goal pane is cleared since this tactic successfully resolves the goal. When all goals are resolved, the proof can be concluded with `qed`. This saves the lemma for future use, and **EasyCrypt** provides confirmation in the response pane as shown:

```
+ added lemma: `int_refl'
```

The `trivial` tactic attempts to solve the goal by applying a variety of internal tactics. While it may sometimes be unclear when it will succeed, the advantage of `trivial` is that it never fails—it either resolves the goal or leaves it unchanged. This makes it a safe and effective tactic to apply without any risk of disruption.

3.1.2 Tactic: `apply`

Once the lemma `int_refl` is proved, **EasyCrypt** stores it and allows us to use it to prove other lemmas. This is accomplished using the `apply` tactic. For example:

```
lemma nineteen_equal: 19 = 19.
proof.
  apply int_refl.
qed.
```

Code 4: Using the `apply` tactic

The `apply` tactic works by attempting to match the conclusion of the provided proof term with the goal's conclusion. If a match is found, the goal is replaced by the subgoals of the proof term.

In this case, **EasyCrypt** matches `int_refl` with the goal's conclusion, verifies the match, and replaces the goal with the subgoals required for `int_refl`. Since there are no additional subgoals to prove for `int_refl`, the proof is concluded successfully.

EasyCrypt also includes a library of predefined lemmas and axioms that can be used to facilitate proofs. For example, `addzC` and `addzA` are axioms related to the commutativity and associativity of integer addition, respectively. We can inspect these predefined lemmas and axioms using the `print` command. For instance, running `print addzC` and `print addzA` prompts **EasyCrypt** to display the following:

```
axiom nosmt addzC: forall (x y : int), x + y = y + x.
```

```
axiom nosmt addzA: forall (x y z : int), x + (y + z) = x + y + z.
```

3.1.3 Tactic: `simplify`

In proofs, it is common for tactics to produce goals that can be simplified. To handle such cases, we use the `simplify` tactic, which reduces the goal to its normal form using principles of lambda calculus. While the underlying mechanics of this process need not concern us, it is crucial to understand that **EasyCrypt** simplifies goals whenever possible, provided it has sufficient knowledge to do so. If the goal is already in normal form, the simplify tactic leaves it unchanged.

For example, the following illustration demonstrates the use of the `simplify` tactic:

```
lemma x_plus_comm (x: int): x + 2*3 = 6 + x.
proof.
  simplify.
  (* EC does the mathematical computation for us and simplifies the goal *)
  simplify.
  (* simplify doesn't fail, and leaves the goal unchanged *)
  trivial.
  (* trivial doesn't fail either, and leaves the goal unchanged *)
  apply addzC.
  (* Discharges the goal *)
qed.
```

Code 5: Using the `simplify` tactic

```
Current goal
Type variables: <none>

x: int
----------------------
x + 2 * 3 = 6 + x
```

$\xrightarrow{\text{simplify}}$

```
Current goal
Type variables: <none>

x: int
----------------------
x + 6 = 6 + x
```

$\xrightarrow{\text{simplify}}$

```
Current goal
Type variables: <none>

x: int
----------------------
x + 6 = 6 + x
```

3.1.4 Tactics: move, rewrite, assumption

Until now, we have worked with lemmas that did not involve any assumptions, apart from specifying variable types. However, in most cases, we will need to incorporate assumptions about variables into our proofs. These assumptions are treated as given and are introduced into the context using the `move =>` command, followed by the desired name for the assumption.

When such assumptions appear as goals, rather than explicitly applying them, we can use the `assumption` tactic to discharge the goal directly. This tactic instructs **EasyCrypt** to automatically search for assumptions in the context that match the goal and apply them.

Consider the following example, where we use the axiom `addz_gt0`, which is stated as follows:

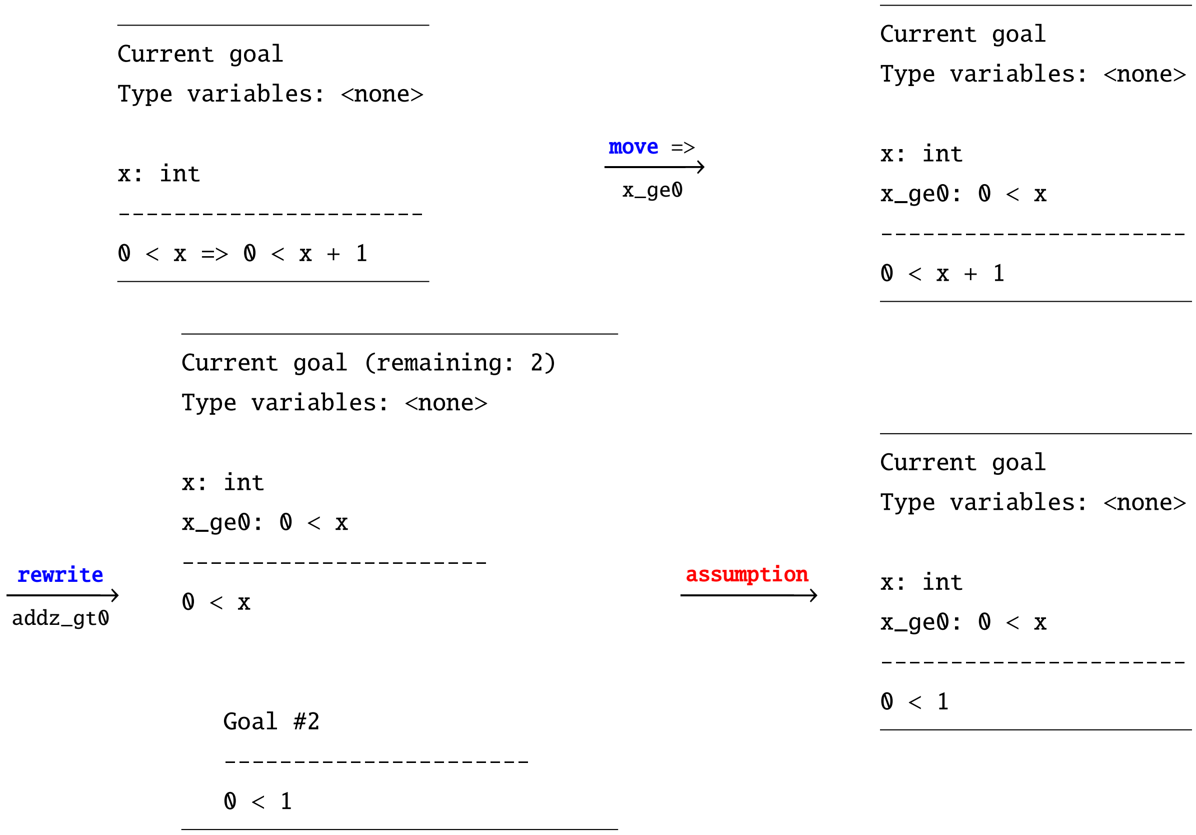
```
axiom nosmt addz_gt0: forall (x y : int), 0 < x => 0 < y => 0 < x + y.
```

Using this result, we construct the following proof script:

```
lemma x_pos (x: int): 0 < x => 0 < x+1.
proof.
  move => x_ge0.
  rewrite addz_gt0.
  (*
  "rewrite" simply rewrites the pattern provided, so in our case it
  rewrites our goal here (0 < x + 1), with the pattern that we provided
  which is addz_gt0, and then requires us to prove the assumptions of
  the pattern which are 0 < x and 0 < 1.
  *)
    (* Goal 1: 0 < x *)
    assumption.
    (* Goal 2: 0 < 1 *)
    trivial.
qed.
```

Code 6: Proof for `x_pos`

In this proof, the `rewrite` tactic modifies the goal by replacing the current expression with a specified pattern. For example, in this case, the goal `0 < x + 1` is rewritten using the pattern from `addz_gt0`, resulting in subgoals that require proving the assumptions `0 < x` and `0 < 1`.



```
Current goal
Type variables: <none>

x: int
----------------------
0 < x => 0 < x + 1
```

```
Current goal
Type variables: <none>

x: int
x_ge0: 0 < x
----------------------
0 < x + 1
```

```
Current goal (remaining: 2)
Type variables: <none>

x: int
x_ge0: 0 < x
----------------------
0 < x

   Goal #2
   ----------------------
   0 < 1
```

```
Current goal
Type variables: <none>

x: int
x_ge0: 0 < x
----------------------
0 < 1
```

Sometimes, a lemma or axiom may be rewritten to the goal, but the left-hand side (LHS) and right-hand side (RHS) of the expression might be flipped. To address this, we can rewrite the lemma or axiom in reverse by prefixing the lemma with a '-' symbol. This approach enables rewriting the sides as follows:

```
lemma int_assoc_rev (x y z: int): x + y + z = x + (y + z).
proof.
  print addzA.
  rewrite -addzA.
  trivial.
qed.
```

Code 7: Rewriting in reverse

These tactics constitute the foundational tools for theorem proving in **EasyCrypt**, particularly when working at the level of ambient logic.

It is worth noting that these tactics, while simple, include a variety of options and intricacies. For instance, the `move =>` tactic supports many introduction patterns, and the keyword move can be substituted with other tactics depending on the context. Expanding on these introduction patterns with clear examples would be a valuable addition to this discussion on basic tactics.

3.1.5 Commands: search and print

When working with theorems, it is often necessary to search through results already available in the environment. While we have encountered a few examples of printing in the content covered so far, it is useful to take a more detailed look at this aspect of **EasyCrypt**, as it is a feature we frequently rely on.

The print command outputs the requested information in the response pane. This command can be used to print various elements, such as types, modules, operations, and lemmas, by using the print keyword. For example:

```
print op (+).
(* abbrev (+) : int -> int -> int = CoreInt.add. *)
print op min.
(* op min (a b : int) : int = if a < b then a else b. *)
print axiom Int.fold0.
(* lemma fold0 ['a]: forall (f : 'a -> 'a) (a : 'a), fold f a 0 = a. *)
```

Code 8: Using the print command

Keywords serve as qualifiers and filters to refine the results. However, it is not mandatory to use qualifiers; printing without them will display broader results, while qualifiers help narrow the scope of the output.

The search command allows us to locate axioms and lemmas involving specific operators. This command takes arguments enclosed in braces to perform the search:

1. [] - Square brackets for unary operators
2. () - Round brackets for binary operators
3. Names of operators
4. Combination of these separated by a space

```
search [-].
search (+).
search ( * ).

search min.
(* Shows lemmas and axioms that include the operator "min". *)

search (+) (=) (=>).
(* Shows lemmas and axioms which have all the listed operators. *)
```

Code 9: Using the search command

3.1.6 External solvers: smt

It is essential to understand that **EasyCrypt** (**EC**) was primarily designed to handle cryptographic properties and more complex constructs. While it is possible to prove general mathematical theorems and claims in **EC**, the process can be cumbersome. To address the challenge of low-level logic and tactics, **EC** integrates with powerful automated tools through the `smt` tactic.

When the `smt` tactic is invoked, **EC** sends the current goal and its context to external **SMT** solvers, such as **Z3** and **Alt-ergo**, which are pre-configured for use with **EC**. If the **SMT** solver can resolve the goal, the `smt` tactic automatically discharges the specific subgoal. However, if the solver fails to find a solution, the proof remains incomplete, and the responsibility of resolving the goal falls back on the user.

For instance, consider the following results:

$$\forall x \in \mathbb{R},\ \forall a, b \in \mathbb{Z},\ x^{a \cdot b} = (x^a)^b,$$
$$\forall x \in \mathbb{R},\ \forall a, b \in \mathbb{Z},\ x \neq 0 \implies x^a \cdot x^b = x^{a+b}.$$

These can be proved in EC using the following script:

```
lemma exp_prod (x: real) (a b: int):
  x^(a*b) = x ^ a ^ b.
proof.
  search (^) ( * ) (=).
  by apply RField.exprM.
qed.

lemma exp_prod2 (x: real) (a b: int):
  x <> 0%r
  => x^a * x^b = x^(a + b).
proof.
  move => x_pos.
  search (^) (=).
  print  RField.exprD.
  rewrite -RField.exprD.
    assumption.
  trivial.
qed.
```

Code 10: Manual proof for `exp_prod` and `exp_prod2`

Alternatively, the proof can be simplified to:

```
lemma exp_prod_smt (x: real) (a b: int): x^(a*b) = x ^ a ^ b.
proof.
  smt.
qed.

lemma exp_prod2_smt (x: real) (a b: int): x <> 0%r => x^a * x^b = x^(a + b).
proof.
  smt.
qed.
```

Code 11: Using `smt` to prove `exp_prod_smt` and `exp_prod2_smt`

The key takeaway is that we will depend heavily on external solvers to handle a significant portion of the computational workload, particularly for results involving low-level mathematics.

This chapter concludes with an overview of ambient logic. We covered fundamental tactics such as **`apply`**, **`simplify`**, **`move`**, and **`rewrite`**, along with the usage of `search` and `print` commands. Additionally, we explored how to work with external solvers to streamline the proving process. In subsequent chapters, we will introduce more advanced tactics and techniques to expand on this foundation.

4 Hoare Logic

Prior sections of this work focused exclusively on deductive proofs operating within purely logical domains. However, the analysis of computational systems—such as algorithms and procedural implementations—demands methodologies capable of formalizing and verifying program semantics.

Consider, for example, an integer exponentiation procedure defined as follows:

$$\exp(a, n) = \begin{cases} 1 & \text{if } n = 0 \\ a \times \exp(a, n-1) & \text{otherwise} \end{cases}$$

Code 12: Pseudo code for exponentiation

```
exp(a, n):
   r = 1
   i = 0
   while (i < n):
      r = r * a
      i = i + 1
   return r
```

When evaluating such a procedure, our objective is to establish its behavioral correctness relative to mathematical specifications. Superficially, this implementation appears valid. However, a critical defect arises: for any negative integer n, the procedure unconditionally returns 1, which contravenes the expected behavior of exponentiation over integers. Consequently, asserting the correctness of this implementation constitutes an invalid claim. To formally characterize program behavior, we instead frame assertions in terms of mathematical invariants and pre/postconditions. For instance:

$$\forall a \in \mathbb{Z},\ n \in \mathbb{Z}_{\geq 0},\ \exp(a, n) = a^n.$$

In other words,

$$\text{Given } \underbrace{x \in \mathbb{Z}, n \in \mathbb{Z}, n \geq 0}_{\text{pre-condition}},\ \underbrace{\exp(a, n)}_{\text{program}} \text{ returns } \underbrace{r = a^n}_{\text{post-condition}}.$$

4.1 Preliminaries: States, Assertions, and Commands

Let $\mathcal{V}$ be a countable set of *program variables* and let $\mathcal{D}$ be a nonempty domain (e.g., $\mathbb{Z}$, $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$, or more structured data). A **state** is a total function

$$s : \mathcal{V} \to \mathcal{D}.$$

We denote by Σ the set of all states. For $x \in \mathcal{V}$ and $d \in \mathcal{D}$, the notation

$$s[x \mapsto d]$$

refers to the state identical to s except that x is updated to d.

Let $\mathcal{L}$ be a logical language (e.g., first-order logic) whose formulas—called **assertions**—are interpreted over Σ. We write $s \models P$ to indicate that the assertion P holds in state s.

We assume a set $\mathcal{E}$ of arithmetic and Boolean expressions over $\mathcal{V}$ (with the usual interpretations). For $E \in \mathcal{E}$, we denote by $[\![E]\!](s) \in \mathcal{D}$ its value in state s.

4.2 Syntax and Semantics of Commands

[Syntax] The syntax of **commands** C is defined inductively by:

$$\begin{array}{rcll} C & ::= & \texttt{skip} & \\ & \mid & x := E & \text{assignment} \\ & \mid & C_1; C_2 & \text{sequencing} \\ & \mid & \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 & \text{conditional} \\ & \mid & \textbf{while } b \textbf{ do } C & \text{iteration} \end{array}$$

where $x \in \mathcal{V}$, $E \in \mathcal{E}$, and b is a Boolean expression.

[Semantic] We define the (partial) **big-step semantics** relation

$$\langle C, s \rangle \Downarrow t \text{ (or } C : s \Downarrow t),$$

which relates an initial state s to a final state t after executing command C. The semantics is given by the following inductive clauses:

1. **Skip**:

$$\langle \texttt{skip}, s \rangle \Downarrow s.$$

2. **Assignment**:

$$\langle x := E, s \rangle \Downarrow s[x \mapsto [\![E]\!](s)].$$

3. **Sequencing**:

$$\frac{\langle C_1, s\rangle \Downarrow s' \quad \langle C_2, s'\rangle \Downarrow s''}{\langle C_1; C_2, s\rangle \Downarrow s''}.$$

4. **Conditional**:

$$\frac{s \models b \quad \langle C_1, s\rangle \Downarrow s'}{\langle \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2, s\rangle \Downarrow s'} \quad \frac{s \not\models b \quad \langle C_2, s\rangle \Downarrow s'}{\langle \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2, s\rangle \Downarrow s'}.$$

5. **While**:

$$\frac{s \not\models b}{\langle \textbf{while } b \textbf{ do } C, s\rangle \Downarrow s}, \quad \frac{s \models b \quad \langle C, s\rangle \Downarrow s' \quad \langle \textbf{while } b \textbf{ do } C, s'\rangle \Downarrow s''}{\langle \textbf{while } b \textbf{ do } C, s\rangle \Downarrow s''}.$$

A command C is said to **terminate** on state s if there exists some t with $\langle C, s\rangle \Downarrow t$.

4.3 Hoare Triples

Hoare Triple

Definition 1. A **Hoare triple** is an expression of the form

$$\{P\}\ C\ \{Q\},$$

where:

- P and Q are assertions, i.e., Boolean-valued formulas over states in $\mathcal{V}$. Formally, P, Q are predicates
$$P, Q : \mathcal{D}^{\mathcal{V}} \to \{\texttt{true}, \texttt{false}\}$$
- C is a command in the syntax given above.

The intended meaning is that if the precondition P holds in the initial state, and if C terminates, then the postcondition Q holds in the final state.

Remark 1. We say $\{P\}\, C\, \{Q\}$ is **valid (or partially correct)** if and only if for all states $s \in \mathcal{D}^{\mathcal{V}}$,

- If $s \models P$ (i.e., P holds in state s), and
- If $\langle C, s\rangle \Downarrow t$ for some state t (i.e., C terminates on s),

then $t \models Q$ (i.e., Q holds in the final state t). We denote validity by $\models \{P\}\, C\, \{Q\}$.

4.4 HL in EasyCrypt

4.4.1 Basic Hoare Triples

We introduce a module that defines two procedures for the programs.

```
module Func1 = {
  proc add_1 (x: int) : int = { return x + 1; }
  proc add_2 (x: int) : int = { x <- x + 2; return x; }
}.
```

Code 13: Function definitions (`Func1`)

A Hoare triple of the form $\{P\}\, C\, \{Q\}$, which is conventionally read as "if P holds prior to executing C, then Q holds afterward," is expressed in **EasyCrypt** as

`hoare [C : P ==> Q]`

in line with the usual definitions. Moreover, **EasyCrypt** records the return value of the program C in a special keyword called `res`. For instance, the Hoare triple

$$\{x = 1\}\, \texttt{Func1.add\_1}\, \{x = 2\}$$

is represented in **EasyCrypt** as follows:

```
lemma triple1: hoare [ Func1.add_1 : x = 1 ==> res = 2].
```

When working within Hoare logic or its variants, the proof goal typically takes a different form than that of ambient logic. For example, after invoking the `lemma triple1`, the resulting goal in **EasyCrypt** 's proof state exhibits precisely these Hoare-specific features.

```
Type variables: <none>

------------------------------------------------------------------------
pre = arg = 1
   Func1.add_1
post = res = 2
```

Code 14: Goal upon evaluating (`triple1`)

To advance the proof, we must inform **EasyCrypt** about the definition of `Func1.add_1`. This is accomplished by the `proc` tactic, which incorporates the procedure's body into the current proof context. Since `Func1.add_1` only returns a value, applying **proc** replaces **res** with that return value and effectively leaves the program body empty for further analysis.

When reasoning about program correctness using Hoare Logic, we systematically analyze how program statements transform preconditions into postconditions, guided by axioms and lemmas. The process involves "consuming" program statements—applying rules to decompose the program until no statements remain. At this point, the goal transitions from a Hoare Logic (HL) goal to a statement in ambient logic, facilitated by the **skip** tactic.

Specifically, skip applies the following reasoning:

$$\frac{P \implies Q}{\{P\}\,\text{skip};\,\{Q\}}\ \texttt{skip}$$

where skip; denotes an empty program, and **skip** is the tactic that enacts this transition. As a result, we return to the standard ambient logic setting, allowing us to use all the tactics. The only subtlety is that moving from a Hoare logic goal to an ambient logic goal requires accounting for the memory qualifiers in which the program variables reside.

For instance, in this example, after applying **skip**, the resulting goal is:

```
forall &hr, x{hr} = 1 => x{hr} + 1 = 2,
```

where `&hr` denotes a particular memory state, and `x{hr}` is the value of `x` within that memory. Proving this is straightforward: we simply move the memory parameter into our assumptions by prepending the `&` character in the **move** `=>` tactic.

Putting these steps together yields the following complete proof for our simple example:

```
lemma triple1: hoare [ Func1.add_1 : x = 1 ==> res = 2].
proof.
  proc.
  skip.
  move => &m H1. (* &m moves memory to the environment *)
  subst. (* Substitutes variables from the assumptions *)
  trivial.
qed.
```

4.4.2 Automation, and Special Cases

4.4.3 Conditionals and Loops

4.5 Hoare Logic with AES

We describe the specifications, the corresponding Hoare triples, and the formal proof strategies used to show that the imperative specifications (for key expansion and AES rounds) are equivalent to their functional counterparts.

4.5.1 Overview

We consider an implementation of the AES algorithm that consists of several imperative procedures. In particular, we verify the following:

1. **Key Expansion.**

   The procedure ‘`Aes.keyExpansion`‘ is specified by the Hoare triple

   $$\{\,\texttt{key} = k\,\}\ \texttt{Aes.keyExpansion}\ \{\,\forall i,\ (0 \le i < 11 \implies \texttt{res}[i] = \texttt{key\_i}\ k\ i)\,\}.$$

   That is, when invoked with an initial key k, the result (an array of round keys) satisfies, for each i with $0 \le i < 11$, the equation $\texttt{res}[i] = \texttt{key\_i}(k, i)$.

   ```
   hoare aes_keyExpansion k :
     Aes.keyExpansion : key = k
     ==>
     forall i, 0 <= i < 11 => res.[i] = key_i k i.
   ```

2. **AES Rounds.**

   The procedure ‘`Aes.aes_rounds`‘ is specified by the Hoare triple

   $$\{\,(\forall i\,(0 \le i < 11 \implies \texttt{rkeys}[i] = \texttt{key\_i}\ k\ i))\wedge(\texttt{msg} = m)\,\}\ \texttt{Aes.aes\_rounds}\ \{\,\texttt{res} = \texttt{aes}\,k\,m\,\}.$$

   That is, given that the round keys are correctly computed (as indicated by the invariant $\forall i,\ (0 \le i < 11 \implies \texttt{rkeys}[i] = \texttt{key\_i}\ k\ i)$) and that the message is m, the final result equals the functional specification $\texttt{aes}\,k\,m$.

   ```
   hoare aes_rounds k m :
     Aes.aes_rounds : (forall i, 0 <= i < 11 => rkeys.[i] = key_i k i) /\ msg = m
     ==>
     res = aes k m.
   ```

3. **Full AES Procedure.**

   The top-level procedure ‘Aes.aes‘ is then specified by the Hoare triple

   $$\{\,(\texttt{key} = k) \wedge (\texttt{msg} = m)\,\}\ \texttt{Aes.aes}\ \{\,\texttt{res} = \texttt{aes}\,k\,m\,\}.$$

In other words, when provided with key k and message m, the AES implementation computes the correct result.

```
hoare aes_h k m :
  Aes.aes : key = k /\ msg = m
  ==>
  res = aes k m.
```

4.5.2 Formal Specification and Proofs

In our formalism, each Hoare triple is written as

$$[P : C \Longrightarrow Q]\,,$$

where P is the precondition, C is the command (or procedure), and Q is the postcondition. The following are the specifications, along with the outlines of their corresponding formal proofs.

4.5.3 Formal Proof Structure

References

[1] Rosulek, Mike. *The Joy of Cryptography*. 2019.

[2] Shah, Tejas Anil. *The Joy of EasyCrypt*. Master's thesis, University of Tartu, 2022.

A Cartesian Closed Category (CCC)

Cartesian Closed Category (CCC)

Definition. A **cartesian closed category (CCC)** is a category $\mathcal{C}$ that has

(i) A **terminal object** 1.

(ii) **Finite products**: For each pair of $X, Y \in \mathrm{Obj}(\mathcal{C})$, there is a product object $X \times Y$.

(iii) **Exponential objects** (or **function spaces**): For each pair of $X, Y \in \mathrm{Obj}(\mathcal{C})$, there is an object Y^X (written also as $\mathrm{Hom}_{\mathcal{C}}(X, Y)$ in some texts) together with the usual adjunction isomorphism $\mathrm{Hom}_{\mathcal{C}}(Z \times X, Y) \simeq \mathrm{Hom}_{\mathcal{C}}(Z, Y^X)$.

We can think of these conditions as ensuring that $\mathcal{C}$ behaves like the category of sets: you can form products of objects (analogs of Cartesian products of sets) and you can also form exponentials (analogs of set functions, i.e., function spaces).

When we talk about the **type system** of a functional programming language (or a proof assistant, or a proof language like EasyCrypt), we usually consider:

1. **Base types** (sometimes called "atomic" or "ground" types), e.g. `unit`, `int`, `bool`, `real`, etc.

2. **Product types** $t_1 * t_2$, which correspond to ordered pairs whose first component has type t_1 and second component has type t_2.

3. **Function types** $t_1 \to t_2$, which correspond to (total) functions that take an input of type t_1 and produce an output of type t_2.

We form a category $\mathcal{C}$ where:

- **Objects** are the various types.

- Morphisms $\sigma : t_1 \to t_2$ are the terms (or expressions) that map values of type t_1 to values of type t_2.

Then

- **Terminal object**: The type `unit` is the *terminal object* because there is exactly one element of type (the value `()`), and thus exactly one morphism from any type t to `unit`.

- **Finite products**: The *product type* $t_1 * t_2$ is the categorical product in C. In particular, there are *natural projection morphisms*

  $$\pi_1 : t_1 * t_2 \longrightarrow t_1, \quad \pi_2 : t_1 * t_2 \longrightarrow t_2,$$

  and for any type u with morphisms $f : u \longrightarrow t_1$ and $g : u \longrightarrow t_2$, there is a unique morphism

  $$\langle f, g \rangle : u \longrightarrow t_1 * t_2$$

  that factors through those projections.

- **Exponentials (function spaces)**: The function type $t_1 \longrightarrow t_2$ is the *exponential object* of t_2 by t_1. Concretely, the usual *currying* isomorphism in functional programming

  $$(t_1 * t_2) \longrightarrow t_3 \simeq t_1 \longrightarrow (t_2 \longrightarrow t_3)$$

  matches the CCC axiom of having an isomorphism

  $$\mathrm{Hom}(Z \times X, Y) \simeq \mathrm{Hom}(Z, Y^X),$$

  capturing that you can either view a function of two arguments as taking a pair (x, y) in one go, or equivalently as taking x first and then returning a function of y.

Therefore, the presence of a terminal type `unit`, product types (to serve as categorical products), and function types (to serve as exponentials) means that **the category of types and terms in EasyCrypt is cartesian closed**.

Intuitively,

- "Cartesian" because we can form product types ($\times$) and have a terminal type (like the set with one element).

- "Closed" because we also have an internal notion of function space, i.e., the type $t_1 \longrightarrow t_2$.

This structure underlies the standard interpretation of simply typed λ-calculus, higher-order logic, and functional programming languages. In short, once you have "pair types" and "function types" in a well-behaved way, you get a cartesian closed category.