The EasyCrypt tool

Documentation and User's Manual

Version 0.2, February 26, 2013

Gilles Barthe[1]
Benjamin Grégoire[2]
Juan Manuel Crespo[1]
César Kunz[1,4]
Santiago Zanella-Béguelin[3]

[1] IMDEA Software Institute, Spain
[2] INRIA Sophia Antipolis, France
[3] Microsoft Research, United Kingdom
[4] Universidad Politécnica de Madrid, Spain

Foreword

This is the manual for the EasyCrypt framework for computer-aided cryptographic proofs. EasyCrypt is an automated tool that supports the machine-checked construction and verification of security proofs of cryptographic systems, and that can be used to verify public-key encryption schemes, digital signature schemes, hash function designs, and block cipher modes of operation.

Availability

EasyCrypt web page can be found at `http://http://easycrypt.gforge.inria.fr/`. Instructions for accessing the source code, documentation, and examples can be found there, together with contact information and recent publications.

See the file `README` for installation instructions.

Contact

There is a public mailing list for users' discussions:

`http://lists.gforge.inria.fr/mailman/listinfo/easycrypt-club`.

Report any bug to the EasyCrypt Bug Tracking System:

`https://gforge.inria.fr/tracker/?atid=8938&group_id=2622&func=browse`

Acknowledgements

We gratelly thank the people who contributed to EasyCrypt: Guido Genzone, Daniel Hedin, Sylvain Heraud, Anne Pacalet.

Contents

I An introduction to EasyCrypt 7

1 EasyCrypt language 9
1.1 Basic declarations 9
1.2 Game declarations 10
1.2.1 Probabilistic statements. 10
1.2.2 Function Definition. 11
1.2.3 Adversary Signature and Declaration. 11
1.2.4 Game definition 11

2 Probabilistic Relational Hoare Logic 13
2.1 Foundations 13
2.2 Judgements 13
2.3 Proof process 13
2.4 Tactics 14
2.4.1 Basic Tactics 14
2.4.2 Program Tansformation Tactics 23
2.4.3 Combination of Tactics 27
2.4.4 Automated Tactics 27
2.4.5 by auto, by eager 30
2.4.6 Open equiv goal 30
2.5 Miscellaneous tool directives 30

3 Probability Claims and Computation 33
3.1 Claims using equiv 33
3.2 Claim using same and split 34
3.3 Deducing claim from other claims 35
3.4 Claims by compute 35
3.5 Claims by auto 37
3.6 Admitting claims 37

4 Example: elgamal 39

II Language Reference 45
4.1 Lexical conventions 47
4.2 Type Expressions. 47
4.3 Expressions. 48
4.4 Declarations. 49
4.5 pRHL judgments 50
4.6 Tactics 51
4.7 Probability claims 52
4.7.1 Program 52

Part I

An introduction to EasyCrypt

Chapter 1

EasyCrypt language

1.1 Basic declarations

Types, constants, operators. EasyCrypt provides native basic types such as `unit`, `bool`, `int`, `real`, `bitstring` as well as polymorphic lists `list`, polymorphic maps `map`, product types `*` (infix notation), and `option` types. Abstract types can be declared with statements of the form `type` *type_ident*, as in the following example:

```
type secret_key.
type group.
```

Parametric type declarations are also supported. Type variables start with a `'` symbol:

```
type 'a list.
```

Types synonyms can be declared with declaration of the form `type` *type_ident* `=` *type_exp*, where *type_exp* is built from basic types, type instantiation, and other user-declared types, as in the following example:

```
type secret_key = int.
type pkey = group.
type ciphertext = group * group.
```

Constants are introduced with declarations of the form `cnst` *ident*`:` *type_exp* [*exp*], where *exp* is an optional expression defining the constant. For example, the following declarations introduce constants with identifiers `g` and `empty_map` of types `group` and `('a, 'b) map`, respectively:

```
cnst g : group.
cnst empty_map : ('a, 'b) map.
```

Operators are introduced with declarations of the form `op` *op_ident* `:` *fun_type* [`as` *id*] where the operator *op_ident* can be either an alpha-numerical identifier or a binary operator —which may include extra symbols such as `=`, `<`, `~`, `+`, `%`, and `^` for example— enclosed in square backets. The identifier *gt_int* is required when defining a binary operator enclosed in brackets, and is used as an internal identifier following the syntactic conventions of the tools in which EasyCrypt relies. The signature *fun_type* is defined with the syntax *type_exp* `->` *type_exp*, or `(`$type\_exp_1$`,`...`,`$type\_exp_k$`)` `->` *type_exp*, where *type_exp* stands for type expressions and $type\_exp_1$`,`...`,`$type\_exp_k$ is a possibly empty list of type expressions. For example:

```
op exp : real -> real
```

The first operator is declared as infix and denoted by the symbol `>`. The operator `exp` is a prefix operator. The definition of polymorphic operators is also allowed by the use of type variables, e.g., the `hd` operator defined in the EasyCrypt prelude:

```
op hd : 'a list -> 'a.
```

As well as constants, operators can be defined by an expression using the following syntax:
`op` *op_ident*`(`*params*`)` `=` *exp* [`as` *id*]
notice that the result type is not required in this case. The following are examples of operators defined in the EasyCrypt prelude:

```
op fst(c : 'a * 'b) = let a,b = c in a.
op [>] (x,y:int) = y < x as gt_int.
```

Probabilistic operators. Probability distributions (see random samplings in the definition of probabilistic statements) can be defined by declaring operators with the syntax `pop` *ident* `:` *fun_type* where, as well as in the definition of deterministic operators, the function signature *fun_type* is defined with the syntax *type_exp* `->` *type_exp*, or `(`$type\_exp_1$`,`...`,`$type\_exp_k$`)` `->` *type_exp*, where *type_exp* stands for type expressions and $type\_exp_1$`,`...`,`$type\_exp_k$ is a possibly empty list of type expressions. For example:

```
pop gen_secret_key : int -> secret_key.
```

Logical formulae. Formulae are built from boolean expressions, standard logical connectives, defined predicates, and logical variable quantification. Boolean expressions are built by the application of native or user-defined operators.

Logical formulae must be closed with respect to logical variables. The syntax for universal quantification is of the form:

```
forall (x,y:int,z:real), p(x,y,z)
```

where `p` is a first-order formula and `x,y,z` are logical variables, and similarly with existential quantification (`exists`).

In addition to logical variables, in some contexts, predicates may contain program variables tagged with a `{1}` or `{2}` flag. A formula defining an axiom must contain only logical variables, whereas formulae describing pre and postconditions on a relational judgment (discussed below) usually refers to tagged program variables.

The special notation to specify that the states on the left and right are equal over a subset of variables. For example, one can write `={x,y,z}` to denote the equivalent relational predicate

```
x{1}=x{2} && y{1}=y{2} && z{1}=z{2}
```

Predicates. Predicates are introduced with the syntax `pred` *ident*`(`*params*`)=` *p* where *params* is a list of formal argument declarations and *p* is a first-order non-relational formula. For example:

```
pred injective(T:('a, 'b) map) =
  forall (x,y:'a), in_dom(x,T) => in_dom(y,T) => T[x] = T[y] => x = y.
```

Axioms and Lemmas. Axioms are used to describe properties of abstract operators and types, or to introduce hypotheses over declared constants. Axioms are defined by a declaration of the form `lemma` *ident* `:` *p*, where *ident* is a valid identifier and *p* is a first-order non-relational formula. For example:

```
axiom head_def : forall (a: 'a, l: 'a list),  hd(a::l) = a.

axiom empty_in_dom : forall (a:'a), !in_dom(a, empty_map).
```

The axiom `head_def` defines the list operator `hd`. The axiom `empty_in_dom` characterizes `empty_map` as a map with an empty domain.

Lemmas can also be introduced to facilitate the verification of later goals. The syntax is similar to the one of axioms: `lemma` *ident* `:` *p*, where *p* is a first-order non-relational formula. When a lemma statement is found, EasyCrypt proves it by calling the available provers/SMT provers through the Why3 tool.

1.2 Game declarations

Games are defined by three components: variables describing the global state, defined procedures and abstract adversary declarations.

1.2.1 Probabilistic statements.

Statements are defined as a list, possible empty, of basic instructions (assignments and function calls) ending on a semicolon, or composed instructions (conditional and while loops). No semicolon is accepted after a conditional or loop statement. Conditional statements follow the syntax `if (`*b*`) {` *stmt* `}` where *stmt* is a probabilistic statement and *b* is a boolean guard. While loop statements follow the syntax `while (`*b*`) {` *stmt* `}`. Curly brackets are not required when *stmt* contains a single instruction.

Probabilistic assignments are of the form `ident =` *d_exp* where *d_exp* is a probability expression, such as uniform distributions over booleans (`{0,1}`), integer intervals `[i..j]`, and bitstrings of arbitrary length (`{0,1}^k`), or distributions defined in terms of probabilistic operators. Assume `gen_secret_key : int -> secret_key` is a defined probabilistic operator, the following are valid probabilistic assignments:

```
x = {0,1}
x = [0..q-1]
x = {0,1}^k
x = gen_secret_key(0)
```

1.2.2 Function Definition.

Functions are defined either by a function body containing variable declarations and probabilistic statements or as synonyms of functions of already defined games.

- `fun` *fun_ident* `(`*typed_args*`) :` *ret_type* `= {` *fun_body* `}`

  fun_ident is a valid function identifier, a list of typed formal parameters *typed_args*, the return type *ret_type* and its body *fun_body*. The function body is defined as a list of local variable declarations of the form `var` *ident* `:` *type*`;`, a probabilistic statement, and a return instruction of the form `return` *exp*, where *exp* is a deterministic expression.

- `fun` *fun_ident* `=` *game_ident*`.`*fun_ident*

  The resulting function has the same formal parameters and function body than the function on the right.

1.2.3 Adversary Signature and Declaration.

Adversary signatures are defined outside a game declaration with a syntax of the form:
`adversary` *adv_sign_ident*`(`*typed_args*`) :` *res_type* `{`o_sign_1`,...,`o_sign_k`}.`
where *res_type* is a type expression specifying the return type and $o\_sign_1,\ldots,o\_sign_k$ is a list (possibly empty), of oracle signatures. In the following example

```
adversary A1_sign(pk:pkey)  : message * message { group -> message}.
adversary A2_sign(c:cipher) : bool              { group -> message}.
```

the type expressions `message*message` and `bool` indicate the return type. A list of signatures in square brackets indicates the signature of the oracles that can be invoked by adversaries with these signatures. In this particular example both signatures belong to adversaries that can invoke a single oracle with type `group -> message`.

As well as function definition, adversaries are either declared abstractly or as adversary synonyms. Abstract declarations follow the syntax:
`abs` *adv_ident* `=` *adv_sign_ident* `{` $ident_1,\ldots,ident_k$`}`
For the adversary signature above we can write for example:

```
abs A1 = A1_sign {H_A}
abs A2 = A2_sign {H_A}
```

where `H_A` is a defined function representing an oracle. Clearly, EasyCrypt requires the function `H_A` to have the signature `group -> message`.

Adversary synonyms follow a similar syntax to function synonyms:
`fun` *adv_ident* `=` *game_ident*`.`*adv_ident*
The result of this declaration is, however, not necessarily an abstract adversary.

1.2.4 Game definition

- A game can be defined by the following syntax:

  Syntax `game` *ident* `= {`*game_body*`}` The body of a game *game_body* is composed of a global variable declaration, function definitions and abstract adversary declarations. The declaration of global variables consists of a list of statements of the form `var` *ident* `:` *type* as in the definition of function local variables, except that they are not separated by a semicolon.

- Alternatively, one can redefine a game by removing or adding variables, and redefining functions from an already defined game.

Syntax `game` *ident* `=` *g_ident var_modifs*
`where` $ident_1$ `= {` *fun_body* `} and` ... `and` $ident_k$ `= {` *fun_body* `}`.

The *g_ident* identifier refers to an existing game, *var_modifs* consists of an optional statement of the form `remove` $ident_1$,..,$ident_k$ and a possible empty list of new variable declarations. Finally, a list of function redefinitions is given separated by the `and` keyword.

Chapter 2

Probabilistic Relational Hoare Logic

2.1 Foundations

Probabilistic Relational Hoare Logic (pRHL) judgments are quadruples of the form:

$$\models c_1 \sim c_2 : \Psi \Longrightarrow \Phi$$

where c_1, c_2 are programs and Ψ, Φ are first-order relational formulae. Relational formulae are first-order formulae over logical variables and program variables tagged with either `{1}` or `{2}` to denote their interpretation in the left or right-hand side program. The special keyword `res` denotes the return value of a procedure and can be used in the place of a program variable. One can also write `e{i}` for the expression —e— in which all program variables are tagged with `{i}`. A relational formula is interpreted as a relation on program memories. See the related articles [?] for more information on this logic.

2.2 Judgements

In EasyCrypt, pRHL judgments are introduced with judgments of the form

```
equiv Fact : Game1.f1 ~ Game2.f2 : Pre ==> Post.
```

where `Fact` is a judgment identifier, `Game1` and `Game2` are games, `f1` and `f2` are identifiers for procedures in `Game1` and `Game2` respectively. The procedures `f1` and `f2` may be abstract or concrete; however, judgments between two abstract procedures can only be defined only if the two abstract procedures correspond to the same adversary.

The pre-condition `Pre` and post-condition `Post` are relational formulae, and define relations between the parameters and the global variables of the two procedures, the post-condition is a relation between the global variables and a special variable named `res`, representing the return value of the procedures. More precisely `res{1}` stands for return value of the left procedure and `res{2}` stands for the return value of the right procedure. For convenience, EasyCrypt also allows pre-conditions and post-conditions to include sub-formulae of the form `={x1, ..., xn}` stating that the values of `x1 ... xn` coincide in the left and right memories. That is, `={x1, ..., xn}` is a shorthand for `x1{1}=x1{2} && ... && xn{1}=xn{2}`.

EasyCrypt also supports judgments of the form:

```
equiv Fact : Game1.f1 ~ Game2.f2 : (Inv).
```

as a shorthand for

```
equiv Fact : Game1.f1 ~ Game2.f2 : ={params} && Inv  ==>  ={res} && Inv.
```

where `params` is the list of parameters of `f1` and `f2`. Note that in order for the judgment to be meaningful, the procedures must have the same return type and the same signature type.

2.3 Proof process

A statement of the form

```
equiv Fact : G1.f1 ~ G2.f2 : Pre ==> Post.
```

opens a verification process, provided `f1` and `f2` are both abstract procedures, or both concrete procedures.

In case `f1` and `f2` are both abstract procedures, the only available tactic is `auto`. Note that, since abstract procedures are allowed to call concrete procedures, it is sometimes useful to prove invariants on the latter prior to proving equivalence properties on `f1` and `f2`.

In case both procedures `f1` and `f2` are concrete, EasyCrypt automatically transforms the judgment into a judgment on their bodies. The pre-condition remains unchanged, but the post-condition is modified by replacing the variables `res{1}` and `res{2}` by the return expressions of `f1` and `f2` respectively.

For example, in the file `examples/elgamal.ec` after the definition of the game `DDH0` we can start a new judgment, stating that the two procedures `INDCPA.Main` and `DDH0.Main` are equivalent if we observe their results (`={res}` stands for `res{1} = res{2}`):

```
equiv CPA_DDH0 : INDCPA.Main ~ DDH0.Main : true ==> ={res}.
```

The judgment is automatically transformed into the following goal:

```
pre   = true
stmt1 =   1 : (sk, pk) = KG ();
          2 : (m0, m1) = A1 (pk);
          3 : b = {0,1};
          4 : mb = if b then m0 else m1;
          5 : c = Enc (pk, mb);
          6 : b' = A2 (pk, c);
stmt2 =   1 : x = [0..q - 1];
          2 : y = [0..q - 1];
          3 : d = B (g ^ x, g ^ y, g ^ (x * y));
post  = (b{1} = b'{1}) = d{2}
```

At this point, the EasyCrypt interpreter expects the user to provide tactics to guide the verification of the judgment. Each tactic may generate both logical verification goals (first-order formulae) that are sent to SMT solvers and new verification subgoals that are stacked for later verification by the user. The interactive verification task concludes when there are no more goals in the stack and the result is *saved* (by typing `save`) or when the verification goal is *aborted*.

Note that we have not implemented support to reason about the case where one procedure is abstract, and another concrete. One possible workaround is to wrap the abstract procedure, say `f1`, into a concrete procedure `f1c` that simply calls `f1`.

2.4 Tactics

2.4.1 Basic Tactics

2.4.1.1 The `app` tactic

Syntax `app` *num num relational-formula*

Description Applies the RHL rule for sequential composition:

$$\frac{\models c_1 \sim c_2 : \Phi \Longrightarrow \Phi' \qquad \models c_1' \sim c_2' : \Phi' \Longrightarrow \Phi''}{\models c_1; c_1' \sim c_2; c_2' : \Phi \Longrightarrow \Phi''}\text{[R-Seq]}$$

The application of tactic `app m n p` defines c_1 as the first `m` instructions of the program on the left-hand side and c_2 as the first `n` instructions of the program on the right-hand side and Φ' as `p`.

Example The application of the tactic `app 1 1 ={x}` on the left goal, yields the two goals on the right.

```
pre   = true
stmt1 =    1: x = [0..10];
           2: if (x = 10) x = 0;
                 else x = x - 1;
stmt2 =    1: x = [0..10];
           2: if (x = 10) x = 0;
                 else x = x - 1;
post  = x{1} + 22 + 1 = x{2} + 23
```

```
pre   = true
stmt1 =    1: x = [0..10];
stmt2 =    1: x = [0..10];
post  = ={x}


pre   = ={x}
stmt1 =    1: if (x = 10) x = 0;
                 else x = x - 1;
stmt2 =    1: if (x = 10) x = 0;
                 else x = x - 1;
post  = x{1} + 22 + 1 = x{2} + 23
```

2.4.1.2 The `rnd` tactic

Syntax `rnd` [*side*] [*dir*] [`(`*fct*`)` | `(`*fct*`)` `(`*fct*`)`]

where *side* is `{1}` or `{2}` and *dir* is `<<` or `>>` and *fct* is *relational-expr* or *ident* `->` *relational-expr*

Description The application of this tactic supports several variants depending on its optional arguments:

- the optional argument *side* may be used to indicate the application of the one-sided logical rule for random sampling. If missing, then the two-sided rule for random assignment is considered

- the optional argument *dir* indicates whether the random samplings appear at the bottom (`<<`) or at the top (`>>`) of the instructions in the current goal. When this argument is missing, the default option (`<<`) is considered.

- Additionally, for the two-sided case, the `rnd` tactic takes as parameter a representation of a bijective function. If a single function f is given then it is required to be an involution. If a pair of functions f and g are given then g is required to be the inverse of f. When no function is given the identity function is considered.

  Two syntactic forms are currently supported for the representation of the optional function arguments. The recommended form is v`->`e, where e is a relational expression (an expression with `{1}` and `{2}` tags) and v is a valid variable identifier. Alternatively (and for the time being), and only in case the first or last instruction of the right program is an assignment, one can simply give an expression e, in which case the bound variable v is set to the lhs of the assignment.

The application of the `rnd` tactic always expects the expressions at right of and assignment to be a simple random expression, even though the programming syntax allows more complex expressions like `x = [0..10] + 3`, or even multiple samplings of the form `(x,y) = ({0,1},{0,1}^k)`. To deal with random expressions occurring in mode complex constructions one must first make use of the `derandomize` tactic.

One-sided application. The following table describes the result of one-sided application of the `rnd` tactic. In the following, we denote A as the support of sampled distribution. In the particular case of the uniform distribution over the integer interval `[`k_1`..`k_2`]`, the expression $a \in A$ corresponds to the condition k_1`<=`a `&&` a`<=`k_2.

Syntax	Rule
`rnd{1}` or `rnd{1}<<`	$\dfrac{\models c_1 \sim c_2 : \Psi \Longrightarrow \forall a,\ (a \in A \Rightarrow \Phi\left[{}^{a}/_{x\langle 1\rangle}\right])}{\models c_1; x \xleftarrow{\$} d \sim c_2 : \Psi \Longrightarrow \Phi}$
`rnd{2}` or `rnd{2}<<`	$\dfrac{\models c_1 \sim c_2 : \Psi \Longrightarrow \forall a,\ (a \in A \Rightarrow \Phi\left[{}^{a}/_{x\langle 2\rangle}\right])}{\models c_1 \sim c_2; x \xleftarrow{\$} d : \Psi \Longrightarrow \Phi}$
`rnd>>{1}`	$\dfrac{\models c_1 \sim c_2 : \exists z,\ (x\langle 1\rangle \in A \wedge \Psi\left[{}^{z}/_{x\langle 1\rangle}\right]) \Longrightarrow \Phi}{\models x \xleftarrow{\$} d; c_1 \sim c_2 : \Psi \Longrightarrow \Phi}$
`rnd>>{2}`	$\dfrac{\models c_1 \sim c_2 : \exists z,\ (x\langle 2\rangle \in A \wedge \Psi\left[{}^{z}/_{x\langle 2\rangle}\right]) \Longrightarrow \Phi}{\models c_1 \sim x \xleftarrow{\$} d; c_2 : \Psi \Longrightarrow \Phi}$

Example In this simple example, the application of the tactic `rnd{1}` (or equivalently `rnd{1}<<`), yields the goal on the right at the top. An application of the `rnd{e}>>` over the latter returns the right goal at the bottom, which can be easily discharged by the `trivial` tactic:

```
pre   = 0 <= x{1}
stmt1 =   1 : z = [x * x..y];
stmt2 =   1 : z = [y..y];
post  = 0 <= z{1} && z{2} = y{2}
```

```
pre   = 0 <= x{1}
stmt1 =
stmt2 =   1 : z = [y..y];
post  = forall (z : int),
          x{1} * x{1} <= z => z <= y{1} =>
            0 <= z && z{2} = y{2}
```

```
pre   = y{2} <= z{2} && z{2} <= y{2} && 0<=x{1}
stmt1 =
stmt2 =
post  = forall (z : int),
          x{1} * x{1} <= z => z <= y{1} =>
            0 <= z && z{2} = y{2}
```

Two-sided application. The following table describes the two-sided applications of the `rnd` tactic. The expression $\mathsf{bij}(f, g, a)$ stand for the condition $g(f(a)) = a \wedge f(g(a)) = a$ and the $\mathsf{invol}(f, a)$ stands for $f(f(a)) = a$.

Syntax	Rule
`rnd` or `rnd<<`	$\dfrac{\models c_1 \sim c_2 : \Psi \Longrightarrow d = d' \wedge \forall a,\ (a \in A \Rightarrow \Phi\left[{}^{a}/_{x\langle 1\rangle}\right]\left[{}^{a}/_{y\langle 2\rangle}\right])}{\models c_1; x \xleftarrow{\$} d \sim c_2; y \xleftarrow{\$} d' : \Psi \Longrightarrow \Phi}$
`rnd>>`	$\dfrac{\models c_1 \sim c_2 : d = d' \Rightarrow x\langle 1\rangle \in A \wedge x\langle 1\rangle = x\langle 2\rangle \wedge \exists_{x,y}\Psi\left[{}^{x}/_{x\langle 1\rangle}\right]\left[{}^{y}/_{y\langle 2\rangle}\right] \Longrightarrow \Phi}{\models x \xleftarrow{\$} d; c_1 \sim y \xleftarrow{\$} d'; c_2 : \Psi \Longrightarrow \Phi}$
`rnd (`f`),(`g`)`	$\dfrac{\models c_1 \sim c_2 : \Psi \Longrightarrow d = d' \wedge \forall a,\ (a \in A \Rightarrow \mathsf{bij}(f,g,a) \wedge \Phi\left[{}^{a}/_{x\langle 1\rangle}\right]\left[{}^{f\,a}/_{y\langle 2\rangle}\right])}{\models c_1; x \xleftarrow{\$} d \sim c_2; y \xleftarrow{\$} d' : \Psi \Longrightarrow \Phi}$
`rnd (`f`)`	$\dfrac{\models c_1 \sim c_2 : \Psi \Longrightarrow d = d' \wedge \forall a,\ (a \in A \Rightarrow \mathsf{invol}(f,a) \wedge \Phi\left[{}^{a}/_{x\langle 1\rangle}\right]\left[{}^{f\,a}/_{y\langle 2\rangle}\right])}{\models c_1; x \xleftarrow{\$} d \sim c_2; y \xleftarrow{\$} d' : \Psi \Longrightarrow \Phi}$
`rnd>> (`f`,`g`)`	$\dfrac{\models c_1 \sim c_2 : \begin{array}{l} d = d' \Rightarrow \mathsf{bij}(f,g,a) \Rightarrow \\ \quad x\langle 1\rangle \in A \wedge x\langle 1\rangle = x\langle 2\rangle \wedge \exists_{x,y}\Psi\left[{}^{x}/_{x\langle 1\rangle}\right]\left[{}^{y}/_{y\langle 2\rangle}\right] \end{array} \Longrightarrow \Phi}{\models x \xleftarrow{\$} d; c_1 \sim y \xleftarrow{\$} d'; c_2 : \Psi \Longrightarrow \Phi}$
`rnd>> (`f`)`	$\dfrac{\models c_1 \sim c_2 : \begin{array}{l} d = d' \Rightarrow \mathsf{invol}(f,g,a) \Rightarrow \\ \quad x\langle 1\rangle \in A \wedge x\langle 1\rangle = x\langle 2\rangle \wedge \exists_{x,y}\Psi\left[{}^{x}/_{x\langle 1\rangle}\right]\left[{}^{y}/_{y\langle 2\rangle}\right] \end{array} \Longrightarrow \Phi}{\models x \xleftarrow{\$} d; c_1 \sim y \xleftarrow{\$} d'; c_2 : \Psi \Longrightarrow \Phi}$

The `rnd` tactic also accepts (in all its forms) random samplings assigning a tuple of variables or updating a map. The application of the `rnd` tactic in the assignment of multiple variable assignments and map updates requires using the syntax `v->e` for the function parameters.

Example The example below shows the effect of the application of the tactic `rnd (c ^^ m{2})`, the original left goal and the final right goal.

```
pre   = true
stmt1 =    1 : m = M ();
           2 : k_0 = {0,1}^l;
stmt2 =    1 : m = M ();
           2 : c = {0,1}^l;
post  = (k_0{1} ^^ m{1},m{1})
           = (c{2},m{2})
```

```
pre   = true
stmt1 =    1 : m = M ();
stmt2 =    1 : m = M ();
post  = forall (r : bitstring{l}),
           r ^^ m{2} ^^ m{2} = r
           && (r ^^ m{2} ^^ m{2} = r =>
             (r ^^ m{1},m{1}) = (r ^^ m{2},m{2}))
```

Notice the verification condition for the function `(c->c^^m{2})` (written simply `c^^m{2}` since `c` is the assigned variable at the right side): `r ^^ m{2} ^^ m{2} = r`, requiring the function to be an involution.

2.4.1.3 The `case` tactic

Syntax `case` [*side*] : *prog-expr*

Description The case tactic allows to split the proof in two branches, depending of the initial value of an expression. The *side* argument may be used to to indicate the application of the one-sided logical rule. If no argument is provided, then the two-sided rule is used. In this case, the rule requires that the precondition implies the equality of *prog-expr* on the two sides. The tactic corresponds to the following pRHL rules:

Syntax	Rule
`case{1} e`	$\dfrac{\models c_1 \sim c_2 : \Psi \wedge e\langle 1\rangle \Longrightarrow \Phi \qquad \models c_1 \sim c_2 : \Psi \wedge \neg e\langle 1\rangle \Longrightarrow \Phi}{\models c_1 \sim c_2 : \Psi \Longrightarrow \Phi}$
`case{2} e`	$\dfrac{\models c_1 \sim c_2 : \Psi \wedge e\langle 2\rangle \Longrightarrow \Phi \qquad \models c_1 \sim c_2 : \Psi \wedge \neg e\langle 2\rangle \Longrightarrow \Phi}{\models c_1 \sim c_2 : \Psi \Longrightarrow \Phi}$
`case e`	$\dfrac{\begin{array}{c}\models c_1 \sim c_2 : \Psi \wedge e\langle 1\rangle \wedge e\langle 2\rangle \Longrightarrow \Phi \\ \models c_1 \sim c_2 : \Psi \wedge \neg(e\langle 1\rangle \wedge e\langle 2\rangle) \Longrightarrow \Phi\end{array}}{\models c_1 \sim c_2 : \Psi \Longrightarrow \Phi}$

Example The application of `case{1} : x <= y` transforms the goal on the left into the two goals on the right:

```
pre   = ={x,y}
stmt1 =    1: z = (y <= x) ? x : y;
stmt2 =    1: if (x <= y) z = y;
              else z = x;
post  = z{1} = z{2}
```

```
pre   = ={x,y} && x{1} <= y{1}
stmt1 =    1: z = (y <= x) ? x : y;
stmt2 =    1: if (x <= y) z = y;
              else z = x;
post  = z{1} = z{2}

pre   = ={x,y} && !(x{1} <= y{1})
stmt1 =    1: z = (y <= x) ? x : y;
stmt2 =    1: if (x <= y) z = y;
              else z = x;
post  = z{1} = z{2}
```

2.4.1.4 The `if` tactic

Syntax `if` [*side*]

Description Applies the pRHL rule for conditional. If the *side* argument is given then the corresponding one side rule is used, else the two side rule is used. The `if` tactic expects an conditional as first instruction, if it is not the case, the `ifsync` tactic 2.4.1.5 or `cond` tactic 2.4.2.7 can be used.

Syntax	Rule
`if{1}`	$\dfrac{\models c_1; c \sim c' : \Psi \wedge e\langle 1\rangle \Longrightarrow \Phi \qquad \models c_2; c \sim c' : \Psi \wedge \neg e\langle 1\rangle \Longrightarrow \Phi}{\models \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2; c \sim c' : \Psi \Longrightarrow \Phi}$
`if{2}`	$\dfrac{\models c' \sim c_1; c : \Psi \wedge e\langle 2\rangle \Longrightarrow \Phi \qquad \models c' \sim c_2; c : \Psi \wedge \neg e\langle 2\rangle \Longrightarrow \Phi}{\models c' \sim \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2; c : \Psi \Longrightarrow \Phi}$
`if`	$\dfrac{\begin{array}{c}\vdash \Psi \Rightarrow e\langle 1\rangle = e'\langle 2\rangle \\ \models c_1; c \sim c_1'; c' : \Psi \wedge e\langle 1\rangle \wedge e'\langle 2\rangle \Longrightarrow \Phi \\ \models c_2; c \sim c_2'; c' : \Psi \wedge \neg e\langle 1\rangle \wedge \neg e'\langle 2\rangle \Longrightarrow \Phi\end{array}}{\models \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2; c \sim \mathsf{if}\ e'\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2'; c' : \Psi \Longrightarrow \Phi}$

Example The application of the tactic `if` on the goal of the left, yields the two goals on the right.

```
pre   = ={x}
stmt1 =   1: if (x = 10) x = 0;
                else x = x - 1;
          2: x = x + 22;
          3: x = x + 1;
stmt2 =   1: if (x = 10) x = 0;
                else x = x - 1;
          2: x = x + 23;
post  = ={x}
```

```
pre   = ={x} && x{1} = 10 && x{2} = 10
stmt1 =   1: x = 0;
          2: x = x + 22;
          3: x = x + 1;
stmt2 =   1: x = 0;
          2: x = x + 23;
post  = ={x}

pre   = ={x} && && x{1} <> 10 && x{2} <> 10
stmt1 =   1: x = x - 1;
          2: x = x + 22;
          3: x = x + 1;
stmt2 =   1: x = x - 1;
          2: x = x + 23;
post  = ={x}
```

2.4.1.5 The `ifsync` tactic

Syntax

Description

Example

2.4.1.6 The `while` tactic

Syntax `while` [*side*] [*dir*] : *relational-formula* [: *relational-expr*, *relational-expr*]

Description This tactic applies the pRHL verification rules for loops:

- the optional argument *side* can be either `{1}` or `{2}` to indicate the application of one-sided versions of the rule. If missing, the two-sided rule for loops is considered.
- the argument *relational-formula* is mandatory and is used as loop invariant. It can refer to variables in both the left and right programs.
- the pair of relational expressions are required (and accepted only) in the one-sided application of the rule. They are used to prove termination of the while loop; the first one corresponds to a decreasing variant expression and the second one to a lower bound. If no expressions are given in the one-sided case, EasyCrypt tries to infer them.

In the forward version (`>>`) the information that is provided by the precondition about variables that are not modified in the loop body is used as invariant and propagated after the loop. Similarly with the postcondition in the backwards case (`<<`).

Two-sided version.

Syntax `while` [*dir*] : *relational-formula*

Description Applies the two-sided RHL rule for while loops, using the *relational-formula* as loop invariant. The *dir* is used to indicate if the backward rule (`<<`) or the forward rule (`>>`) should be used. If no *dir* argument is given then the backward rule is used. The backward rule require that the last instruction of the each statements are loop instruction (the first for the forward rule).

Syntax	Rule
`while >> :` I	$$\dfrac{\begin{array}{c}\vdash \Psi \Rightarrow I \wedge e\langle 1\rangle = e'\langle 2\rangle \\ \models c_1 \sim c_1' : I \wedge e\langle 1\rangle \wedge e'\langle 2\rangle \wedge \exists M, \Psi \Longrightarrow I \wedge e\langle 1\rangle = e'\langle 2\rangle \\ \models c_2 \sim c_2' : I \wedge \neg e\langle 1\rangle \wedge \neg e'\langle 2\rangle \wedge \exists M, \Psi \Longrightarrow \Phi\end{array}}{\models \mathsf{while}\ e\ \mathsf{do}\ c_1; c_2 \sim \mathsf{while}\ e'\ \mathsf{do}\ c_1'; c_2' : \Psi \Longrightarrow \Phi}$$
`while [<<] :` I	$$\dfrac{\begin{array}{c}\models c_2 \sim c_2' : I \wedge e\langle 1\rangle \wedge e'\langle 2\rangle \Longrightarrow I \wedge e\langle 1\rangle = e'\langle 2\rangle \\ \models c_1 \sim c_1' : \Psi \Longrightarrow I \wedge e\langle 1\rangle = e'\langle 2\rangle \wedge \forall M, (I \wedge \neg e\langle 1\rangle \wedge \neg e'\langle 2\rangle \Rightarrow \Phi)\end{array}}{\models c_1; \mathsf{while}\ e\ \mathsf{do}\ c_2 \sim c_1'; \mathsf{while}\ e'\ \mathsf{do}\ c_1' : \Psi \Longrightarrow \Phi}$$

Example The application of the tactic `while : ={x} && x{1} <= 10` to the left goal, yields the two goals on the right.

```
pre   = ={y}                   pre   = (x{1} < 10 = (x{2} < 10) &&
stmt1 =   1 : x = 0;                   ={x} && x{1} <= 10) && x{1} < 10
          2 : while (x < 10)   stmt1 =   1 : x = x + 1;
                x = x + 1;     stmt2 =   1 : x = x + 1;
stmt2 =   1 : x = 0;           post  = x{1} < 10 = (x{2} < 10) &&
          2 : while (x < 10)           ={x} && x{1} <= 10
                x = x + 1;
post  = ={x,y} && x{1} = 10

                               pre   = ={y}
                               stmt1 =   1 : x = 0;
                               stmt2 =   1 : x = 0;
                               post  = (={x} && x{1} <= 10) &&
                                       x{1} < 10 = (x{2} < 10) &&
                                       (forall (x_L, x_R : int),
                                          x_L < 10 = (x_R < 10) =>
                                          x_L = x_R => x_L <= 10 =>
                                         !x_L < 10 =>
                                         (x_L = x_R && ={y}) && x_L = 10)
```

Example The application of the tactic `while >> : ={x} && x{1} <= 10` to the left goal, yields the two goals on the right, plus a logical verification condition (not shown) that is sent to the available SMT solvers.

```
pre   = x{2}=0 && x{1}=0 && ={y}  pre   = (exists (x_L, x_R : int),
stmt1 =   1 : while (x < 10)                x_R = 0 && x_L = 0 && ={y}) &&
                x = x + 1;                (x{1} < 10 = (x{2} < 10) &&
          2 : y = y + 1;                  ={x} && x{1} <= 10) && x{1} < 10
stmt2 =   1 : while (x < 10)      stmt1 =   1 : x = x + 1;
                x = x + 1;        stmt2 =   1 : x = x + 1;
          2 : y = y + 1;          post  = x{1} < 10 = (x{2} < 10) &&
post  = ={x,y} && x{1} = 10               ={x} && x{1} <= 10


                                  pre   = (exists (x_L, x_R : int),
                                            x_R = 0 && x_L = 0 && ={y}) &&
                                          (x{1} < 10 = (x{2} < 10) &&
                                          ={x} && x{1} <= 10) && !x{1} < 10
                                  stmt1 =   1 : y = y + 1;
                                  stmt2 =   1 : y = y + 1;
                                  post  = ={x,y} && x{1} = 10
```

One-sided version.

Syntax `while` *side* [*dir*] : *relational-formula* [: *variant*, *bound*]

Description Applies the one-sided pRHL rule for while loops, using *relational-formula* as loop invariant, *variant* as the decreasing variant (a *relational-term*) and *bound* as the lower bound (a *relational-term*). The variant and the bound are used to check for termination. If no variant and bound are given, EasyCrypt tries to infer them automatically. The forward and backward one-sided rules are described in the table below; only the left (`{1}`) variants are shown; the right (`{2}`) variants are symmetric. In the table, the expressions $\forall X,\ \varphi$ and $\exists X,\ \varphi$ denote, respectively, universal and existential quantification over the set of variables X modified in the loop body c.

Syntax	Rule
`while{1} : I : v, b`	$$\frac{\vdash I \wedge v \leq b \Rightarrow \neg e \quad \models c \sim \mathsf{skip} : b = B \wedge v = C \wedge e \wedge I \Longrightarrow b = B \wedge v < C \wedge I \quad \models c_1 \sim c_2 : \Psi \Longrightarrow I \wedge \forall X, (I \wedge \neg e \Rightarrow \Phi)}{\models c_1; \mathsf{while}\ e\ \mathsf{do}\ c \sim c_2 : \Psi \Longrightarrow \Phi}$$
`while{1}>> : I : v, b`	$$\frac{\vdash I \wedge v \leq b \Rightarrow \neg e \quad \models c \sim \mathsf{skip} : b = B \wedge v = C \wedge (\exists X,\ \Psi) \wedge e \wedge I \Longrightarrow b = B \wedge v < C \wedge I \quad \models c_1 \sim c_2 : (\exists X,\ \Psi) \wedge \neg e \wedge I \Longrightarrow \Phi}{\models \mathsf{while}\ e\ \mathsf{do}\ c; c_1 \sim c_2 : \Psi \Longrightarrow \Phi}$$

Example Applying the one-sided tactic `while{1} : x{1} <= 10` on the goal below on the left results in the two goals on the right. The goal on the top corresponds to the verification of the loop body, and requires proving that the invariant is preserved, the variant decreases, and the bound remains unchanged. The goal on the bottom corresponds to the verification of the rest of the program, and requires proving that the invariant is established before entering the loop, and that the post-condition holds upon exiting.

```
pre   = ={y}
stmt1 =   1 : x = 0;
          2 : while (x < 10)
                x = x + 1;
stmt2 =   1 : x = 10;
post  = ={x,y} && x{1} = 10
```

```
pre   = (bnd{1} = 0 && 10 - x{1} = vrnt{1})
           && x{1} <= 10 && x{1} < 10
stmt1 =   1 : x = x + 1;
stmt2 =
post  = (bnd{1} = 0 && 10 - x{1} < vrnt{1})
           && x{1} <= 10
```

```
pre   = ={y}
stmt1 =   1 : x = 0;
stmt2 =   1 : x = 10;
post  = x{1} <= 10 &&
          (forall (x_L : int),
             x_L <= 10 => !x_L < 10 =>
                (x_L = x{2} && ={y}) && x_L = 10)
```

Example Similarly, one can invoke first the `sp` tactic to the original goal of the previous example, obtaining the left goal. The `while{1} >> : x{1} <= 10` tactic returns the goals on the right, corresponding to the verification of the loop body and the remaining program statements:

```
pre   = x{2} = 10 && x{1} = 0
           && ={y}
stmt1 =   1 : while (x < 10)
                x = x + 1;
stmt2 =
post  = ={x,y} && x{1} = 10
```

```
pre   = (bnd{1} = 0 && 10 - x{1} = vrnt{1}) &&
          (exists (x_L : int),  x{2} = 10
              && x_L = 0 && ={y}) &&
          x{1} <= 10 && x{1} < 10
stmt1 =   1 : x = x + 1;
stmt2 =
post  = (bnd{1} = 0 && 10 - x{1} < vrnt{1})
           && x{1} <= 10
```

```
pre   = (exists (x_L : int),  x{2} = 10
            && x_L = 0 && ={y}) &&
         x{1} <= 10 && !x{1} < 10
stmt1 =
stmt2 =
post  = ={x,y} && x{1} = 10
```

2.4.1.7 The `call` tactic

Syntax `call` *auto-info*

Description The tactic `call` applies the two-sided Relational Hoare Logic rule for procedure calls. The basic use of the `call` is the following

Syntax	Rule
`call using id`	$$\frac{\texttt{id} : \models f \sim g : \Psi_{fg} \Longrightarrow \Phi_{fg} \qquad \models c_1 \sim c_2 : \Psi \Longrightarrow \Psi_{fg}\left[{}^{\vec{a},\vec{b}}/_{f.\mathsf{params},g.\mathsf{params}}\right] \wedge \forall r_1\ r_2\ M,\ post_{fg} \Rightarrow \Phi\left[{}^{r_1,r2}/_{x\langle 1\rangle,y\langle 1\rangle}\right]}{\models c_1; x \leftarrow f(\vec{a}) \sim c_2; y \leftarrow g(\vec{b}) : \Psi \Longrightarrow \Phi}$$

The tactic checks that the specification named `id` exists and that the two functions used in the specification correspond to the one used in the call instruction of each statements.

Example Assume that we have already proved the following specification

```
equiv f_12 : Gcall1.f ~ Gcall2.f : ={y} && x{2} = 1 ==> P(res{1},res{2})
```

Then the application of the tactic `call using f_12` transform the left goal into the right goal

```
pre   = ={z} && x{2} = 1          pre   = ={z} && x{2} = 1
stmt1 =   1 : w = f (0);          stmt1 =
stmt2 =   1 : w = f (0);          stmt2 =
post  = P(w{1} + 1,w{2} + 1)      post  = x{2} = 1 &&
                                           (forall (res_R, res_L : int),
                                              P(res_L,res_R) => P(res_L + 1,res_R + 1))
```

Sometime EasyCrypt perform some simple automatic simplifications on the post-condition, leading to an equivalent formula. For example the if we use the `=` predicate instead of `P` in the previous example the post-condition become simply `x{2} = 1`.

In general, it is not needed to start by proving a specification for the pair of functions (here `f_12`). It is possible give an invariant which will be used to prove the specification automatically (see section **??**). If the invariant is omitted, it is assumed that the invariant is equality over all common global variables of the two games in the judgment. For example the application of the tactic `call (x{2} = 1)` declare a new specification

```
equiv inferred_Gcall1_f_Gcall2_f_0 : Gcall1.f ~ Gcall2.f :
          ={y} && x{2} = 1 ==> ={res} && x{2} = 1
```

and transform the goal as following:

```
pre   = ={z} && x{2} = 1          pre   = ={z} && x{2} = 1
stmt1 =   1 : w = f (0);          stmt1 =
stmt2 =   1 : w = f (0);          stmt2 =
post  = P(w{1} + 1,w{2} + 1)      post  = x{2} = 1 &&
                                           (forall (res_R : int),
                                              x{2} = 1 => P(res_R + 1,res_R + 1))
```

2.4.1.8 The `unfold` tactic.

Syntax `unfold` $[p_1$, ... , $p_n]$

Description unfolds the definition of provided predicates p_1, ..., p_n in the pre and postcondition. If the list of predicates is empty, every defined predicate is unfolded.

Example Assume we have defined the predicates

```
pred eq(x,y:int) = x=y.
pred geq(x,y:int) = x<=y.
pred gt(x,y:int) = x<y.
```

A call of the tactics `unfold eq` and `unfold` to the goal on the left returns the goals on the right (at the top and bottom respectively).

```
pre   = geq(z{1},z{2}) && !gt(z{1},z{2})      pre   = geq(z{1},z{2}) && !gt(z{1},z{2})
stmt1 =                                       stmt1 =
stmt2 =                                       stmt2 =
post  = eq(z{1},z{2})                         post  = ={z}

                                              pre   = z{1} <= z{2} && !z{1} < z{2}
                                              stmt1 =
                                              stmt2 =
                                              post  = ={z}
```

2.4.2 Program Tansformation Tactics

Some EasyCrypt tactics implement standard program transformations that are commonly used when doing crypto proofs, like function inlining or code motion or when dealing with loops, like loop unrolling. These are described next and examples are provided (pre and postconditions will be omitted when they do not change).

2.4.2.1 The `let` tactic

Syntax `let` [*side*] [*position*] *ident* : *type-expr* = *prog-expr*

Description Add an assignment to a variable, which should be fresh at a given position. The default position is 1. If no side argument is given then apply the transformation to both statements.

Example The application of the tactic `let{1} at 2 w : int = x` 1+ transforms the left goal into the right goal:

```
                                              pre   = true
pre   = true                                  stmt1 =   1: x = 0;
stmt1 =   1: x = 0;                                     2: w = x + 1;
          2: y = 1;                                     3: y = 1;
stmt2 =   1: y = 1;                           stmt2 =   1: y = 1;
          2: x = 0;                                     2: x = 0;
post  = x{1} + y{1} = x{2} + y{2}             post  = x{1} + y{1} = x{2} + y{2}
```

2.4.2.2 The `ifneg` tactic

Syntax `ifneg` [*side*] [*position*]

Description Negate the value of conditional instruction and exchange its branches.

- the optional argument *side* can be either `{1}` or `{2}` and it indicates whether the transformation must be applied in the left or right statement respectively. If missing, the transformation is applied to both statements.
- The optional argument *position* determines which instruction is the target of the transformation. It can be either
  - `at n` applies the transformation on the `if` instruction at position `n`.
  - `at n1, .., np` applies the transformation at positions `n1`, `n2`, . . . , `np`.
  - `last` applies the transformation to the last `if` instruction.
  - If no *position* argument is given the tactic applies the transformation to the first `if` instruction.

Example The tactic `ifneg at 2` transforms the left goal into the right goal

```
pre   = true                                  pre   = true
stmt1 =   1 : x = [0..10];                    stmt1 =   1 : x = [0..10];
          2 : if (x <> 10) x = x - 1;                   2 : if (x = 10) x = 0;
              else x = 0;                                   else x = x - 1;
stmt2 =   1 : x = [0..10];                    stmt2 =   1 : x = [0..10];
          2 : if (x <> 10) x = x - 1;                   2 : if (x = 10) x = 0;
              else x = 0;                                   else x = x - 1;
post  = x{1} + 22 + 1 = x{2} + 23             post  = x{1} + 22 + 1 = x{2} + 23
```

2.4.2.3 The `inline` tactic

Syntax `inline` [*side*] [`P1,..,Pj` | *position*] where *position* is `at n1,..,ni` or `last`.

Description Inline the definition of concrete procedures. The *side* argument indicate if the transformation should be applied on the left statement (`{1}`) or on the right statement (`{2}`), if no *side* argument is provided the transformation is applied on both statement. The second argument indicate which procedure should be inlined.

- In the first variant, the procedure `P1` is inlined first, then the procedure `P2` and so on until the procedure `Pj`. Notice that the order is important.
- In the second variant, the list of the procedure calls to be inlined is specified by giving the positions where they appear in the game. This allows to inline just one call to a procedure.
- If no argument is provided then every concrete procedure is inlined.

Example The effect of the tactic `inline` applied to the left goal yields the right goal.

```
stmt1 = y = [0..100];
        x = f (y);
stmt2 = y = [0..100];
        x = f (y);
```

```
stmt1 =   1 : y = [0..100];
          2 : k = y;
          3 : aux = 1;
          4 : res = 0;
          5 : while (aux <> 0) {
                aux = [0..k];
                res = res + aux;
              }
          6 : x = res;
stmt2 =   1 : y = [0..100];
          2 : k = y;
          3 : aux = 1;
          4 : res = 0;
          5 : while (aux <> 0) {
                aux = [0..k];
                res = res + aux;
              }
          6 : x = res;
```

2.4.2.4 The `swap` tactic

Syntax `swap` [*side*] `[`*num*–*num* `]` | *num*] *num*

Description Moves intructions forwards or backwards whenever it is admissible, otherwise it fails. In general, the transformation is admissible if the swapped instructions are independent. Additionally, EasyCrypt tries to swap two instructions s_1 and s_2 if s_1 is a sequence of assignments over variables that are read by s_2, by doing the appropriate substitutions. For example, althought the two first assignments are not independent in the program on the left, the swap tactic allows the following transformation:

$$x = 2;\ y = 5;\ \mathsf{if}\ 0 \leq x\ \mathsf{then}\ z = f(z + y) \qquad \longrightarrow \qquad \mathsf{if}\ 0 \leq 2\ \mathsf{then}\ z = f(z + 5);\ x = 2;\ y = 5$$

- The optional parameter *side* indicates whether the transformation must be applied to the left (`{1}`) or right (`{2}`) statement. If this argument is missing, both the left and right statement are affected.
- The second optional parameter indicates which block of instructions should be moved.
- The last *num* arguments indicate if the block of instruction should be moved down (if *num* is positive), or up (if *num* is negative).
- `swap [i-j] n` pushes the block of instructions on lines between `i` and `j` of `n` positions down if `n` is positive, and of `n` positions up if `n` is negative.
- `swap i n` is a shortcut for `swap [i-i] n`.
- `swap n` pushes the first instruction `n` positions down, if `n` is positive. Pushes the last instruction `n` positions up if `n` is negative.

Example The effect of the tactic `swap{1} 1` applied to the left goal yields to the right goal.

```
stmt1 =   1: aux = {0,1};
          2: res = {0,1};
stmt2 =   1: res = {0,1};
          2: aux = {0,1};
```

```
stmt1 =   1: res = {0,1};
          2: aux = {0,1};
stmt2 =   1: res = {0,1};
          2: aux = {0,1};
```

2.4.2.5 The `unroll` tactic

Syntax `unroll` [*side*] [*position*]

Description Unrolls one iteration of the specified while loop. The loop unrolling is defined as a transformation of the form

$$\mathsf{while}\ b\ \mathsf{do}\ c \quad\longrightarrow\quad \mathsf{if}\ b\ \mathsf{then}\ c; \mathsf{while}\ b\ \mathsf{do}\ c$$

In contrast to the `condt` tactic, the condition b is not required to hold before the loop entrance.

- the optional argument *side* can be either `{1}` or `{2}` and it indicates whether the transformation must be applied on the left or right statement respectively. If missing, the transformation is applied to both statements.
- The optional argument *position* determines which while loop is the target of the transformation. It can be either
  - `at n` unroll the loop at position `n`.
  - `at n1, .., np` unroll the loop at positions `n1`, `n2`, . . . , `np`.
  - `last` unroll the last loop.
  - If no *position* argument is given the tactic unroll the first loop.

Example The tactic `unroll last` transforms the left goal into the right goal.

```
pre   = ={x}
stmt1 =   1 : x = x + 1;
          2 : while (x <= 10) x = x + 2;
          3 : x = x + 3;
          4 : while (x <= 20) x = x + 4;
          5 : x = x + 5;
stmt2 =   1 : x = x + 1;
          2 : while (x <= 10) x = x + 2;
          3 : x = x + 3;
          4 : while (x <= 20) x = x + 4;
          5 : x = x + 5;
post  = ={x}
```

```
pre   = ={x}
stmt1 =   1 : x = x + 1;
          2 : while (x <= 10) x = x + 2;
          3 : x = x + 3;
          4 : if (x <= 20) x = x + 4;
          5 : while (x <= 20) x = x + 4;
          6 : x = x + 5;
stmt2 =   1 : x = x + 1;
          2 : while (x <= 10) x = x + 2;
          3 : x = x + 3;
          4 : if (x <= 20) x = x + 4;
          5 : while (x <= 20) x = x + 4;
          6 : x = x + 5;
post  = ={x}
```

2.4.2.6 The `splitw` tactic

Syntax `splitw` [*side*] [*position*] : *bool-exp*

Description Splits a while loop into two loops. It replaces the instruction `while (e) { c }` by the two instructions `(while (e && bool-exp) { c }; while (e) { c }`

- If *side* is `{1}` apply the transformation on the left statement. If *side* is `{2}` apply the transformation on the right statement. If no side argument is provided apply the transformation on both statement.
- The optional argument *position* determines which while loop is the target of the transformation. It can be either
  - `at n` unroll the loop at position `n`.
  - `at n1, .., np` unroll the loop at positions `n1`, `n2`, . . . , `np`.
  - `last` unroll the last loop.
  - If no *position* argument is given the tactic unroll the first loop.
- The last argument `bool-exp` is a deterministic program expression.

Example the tactic `splitw at 2: x < 10` transforms the left goal into the right one

```
pre   = ={x}
stmt1 =   1 : x = x + 1;
          2 : while (x <= 10)
                x = x + 2;
          3 : x = x + 3;
stmt2 =   1 : x = x + 1;
          2 : while (x <= 10)
                x = x + 2;
          3 : x = x + 3;
post  = ={x}
```

```
pre   = ={x}
stmt1 =   1 : x = x + 1;
          2 : while (x < 10 && x <= 10)
                x = x + 2;
          3 : while (x <= 10)
                x = x + 2;
          4 : x = x + 3;
stmt2 =   1 : x = x + 1;
          2 : while (x < 10 && x <= 10)
                x = x + 2;
          3 : while (x <= 10)
                x = x + 2;
          4 : x = x + 3;
post  = ={x}
```

2.4.2.7 The `condt` and `condf` tactic

Syntax (`condt` | `condf`) [*side*] [*position*] where *position* is `at n` or `last`.

Description Remove the conditional instruction (a `if` or a `while`) at position *position* to its true branch (`condt`) or its false branch (`condf`). It require to show that the corresponding test is true for `condt` or false of `condf`.

- If *side* is `{1}` apply the transformation on the left statement. If *side* is `{2}` apply the transformation on the right statement. If no side argument is provided apply the transformation on both statement.
- The optional argument *position* determines in which instruction the the transformation should be applied. It can be either
  - `at n` applies the transformation to the conditional instruction at position `n`.
  - `at n1, .., np` applies the transformation to the conditional instructions at positions `n1`, `n2`, . . . , `np`.
  - `last` applies the transformation to the last conditional instruction
  - If no *position* argument is given the tactic applies the transformation to the first conditional instruction

In the general case, the tactic generates two goals. The first one is used to prove that the value of the test as the expected one (true for `condt` and false for `condf`). The second one correspond to the initial statement where the conditional instruction is replaced by the corresponding branch. If the position is 1 then the tactic directly try to prove the first goal. If the position is the last instruction and the corresponding branch is empty (this appear frequently with `condf` if the last instruction is a loop) then the tactic generate only one goal where the original post-condition is extended with the condition on the test.

Example The tactic `condt{1} last` transforms the left goal into the two right goals

```
pre   = ={x} && x{1} <= 9
stmt1 =   1 : x = x + 1;
          2 : while (x <= 10) x = x + 2;
stmt2 =
post  = x{2} <= x{1} + 3
```

```
pre   = ={x} && x{1} <= 9
stmt1 =   1 : x = x + 1;
stmt2 =
post  = x{1} <= 10

pre   = ={x} && x{1} <= 9
stmt1 =   1 : x = x + 1;
          2 : x = x + 2;
          3 : while (x <= 10) x = x + 2;
stmt2 =
post  = x{2} <= x{1} + 3
```

Example The tactic `condf{1} last` transforms the left goal into the right goal

```
pre   = ={x} && 10 < x{1}
stmt1 =   1 : x = x + 1;
          2 : while (x <= 10) x = x + 2;
stmt2 =
post  = x{1} + 3 = x{2}
```

```
pre   = ={x} && 10 < x{1}
stmt1 =   1 : x = x + 1;
stmt2 =
post  = !x{1} <= 10 && x{1} + 3 = x{2}
```

2.4.3 Combination of Tactics

EasyCrypt provides a simple combination mechanism that can be used to build more complex tactics from basic tactics. The following is brief a description of the tactic language of EasyCrypt:

- `idtac`: this tactic always succeeds and has no effect on the current goal.
- $tactic_1$`;`$tactic_2$: this tactic applies first the $tactic_1$ to the current goal and then $tactic_2$ to every subgoal generated by $tactic_1$
- *tactic*`; [` $tactic_1$ `| .. |` $tactic_k$: Applies <tactic>, which must generate exactly k subgoals. Then it applies $tactic_i$ to the i^{th}-subgoal. If a tactic is left unspecified the implicit tactic `idtac` is assumed. For example `condt at 2;[trivial | ]` is equivalent to `condt at 2;[trivial | idtac]`
- `*`*tactic* Prefixing any tactic expression *tactic* with '`*`' results in the tactic being repeatedly applied until it fails.
- `!`n *tactic* Repeats the tactic *tactic* given at most n times.
- `try` *tactic* Tries to apply the tactic given as argument, if it fails, it catches the error.

2.4.4 Automated Tactics

In order to simplify proves, EasyCrypt defines a set of heuristic tactics. In this subsection we provide a description of each of this high level tactics and some examples ilustrating the effects on goals.

2.4.4.1 The `wp` tactic

Syntax `wp` [*pos1 pos2*]

Description Computes the relational weakest-precondition of deterministic, loop and procedure-call free program fragments (i.e. deterministic assignments and conditionals). The tactic processes instructions from bottom to top until a random sampling, a loop or a function call is found. In particular, the computation of the weakest precondition over a conditional instruction is only possible if its branches contain only deterministic assignments or deterministic conditionals.

The optional position parameters *pos1* and *pos2* restricting the range of instructions affected by the tactic application. When given two values k_1 and k_2, the weakest precondition computation stops on the k_1-th instruction of the left statement and on the k_2-th instruction of the right statement.

The application of this tactic fails when no instruction can be processed, e.g., Calling the same tactic `wp` over the last goal returns a failure message.

Example: Assume the predicate `pos(m)` is defined as `forall (b:bool), in_dom(b,m) => 0<=m[b]`, and `k1` and `k2` are positive constants. An invocation of the tactic `wp` over the left goal below returns the goal on the right, i.e., stopping at the random assignments in line 4:

```
pre   = 0 <= x{1} && pos(m{1})
stmt1 =   1 : (x, y) = (x + k1,x + k2);
          2 : z = [x..y];
          3 : b = {0,1};
          4 : m[b] = y;
stmt2 =   1 : (x, y) = (x + k1,x + k2);
          2 : z = [x..y];
          3 : b = {0,1};
          4 : m[b] = y;
post  = pos(m{1}) && ={b}
```

```
pre   = 0 <= x{1} && pos(m{1})
stmt1 =   1 : (x, y) = (x + k1,x + k2);
          2 : z = [x..y];
          3 : b = {0,1};
stmt2 =   1 : (x, y) = (x + k1,x + k2);
          2 : z = [x..y];
          3 : b = {0,1};
post  = pos(m{1}[b{1} <- y{1}]) && ={b}
```

Next, an invocation of `rnd;rnd{1}.` returns the goal on the left below, which can be further process by `wp` returning the goal on the right, which can be discharged by `simpl`:

```
pre   = 0 <= x{1} && pos(m{1})
stmt1 =   1 : (x, y) = (x + k1,x + k2);
stmt2 =   1 : (x, y) = (x + k1,x + k2);
          2 : z = [x..y];
post  = forall (z : int),
          x{1} <= z => z <= y{1} =>
            forall (r : bool),
              pos(m{1}[r <- y{1}])
```

```
pre   = 0 <= x{1} && pos(m{1})
stmt1 =
stmt2 =   1 : (x, y) = (x + k1,x + k2);
          2 : z = [x..y];
post  = (let y_L = x{1} + k2 in
        forall (z : int),
          x{1} + k1 <= z => z <= y_L =>
            forall (r : bool),
              pos(m{1}[r <- y_L]))
```

2.4.4.2 The `sp` tactic

Syntax `sp` [*pos1 pos2*]

Description Implements a strongest postcondition transformer over deterministic, loop free and procedure-call free instructions. The invocation of this tactic processes, from top to bottom, the instructions at the left and right of the goal until a random sampling, a loop, or a procedure call is found. If no instruction can be processed, it returns a failure message.

The partial application of the `sp` tactic with optional arguments *pos1* and *pos2* process at most the first *pos1* instructions at the left and the *pos2* instructions at the right of the current goal.

Example The example verified using `wp` can be dually verified with the `sp` tactic. Assume the predicate `pos(m)` is defined as `forall (b:bool), in_dom(b,m) => 0<=m[b]`, and `k1` and `k2` are positive constants. An invocation of the tactic `sp` over the left goal below returns the goal on the right, i.e., stopping at the random assignments in line 2:

```
pre   = 0 <= x{1} && pos(m{1})
stmt1 =   1 : (x, y) = (x + k1,x + k2);
          2 : z = [x..y];
          3 : b = {0,1};
          4 : m[b] = y;
stmt2 =   1 : (x, y) = (x + k1,x + k2);
          2 : z = [x..y];
          3 : b = {0,1};
          4 : m[b] = y;
post  = pos(m{1}) && ={b}
```

```
pre   = exists (x : int),
          y{2} = x + k2 &&
           x{2} = x + k1 &&
            (exists (x_0 : int),
               y{1} = x_0 + k2 &&
               x{1} = x_0 + k1 &&
               0 <= x_0 && pos(m{1}))
stmt1 =   1 : z = [x..y];
          2 : b = {0,1};
          3 : m[b] = y;
stmt2 =   1 : z = [x..y];
          2 : b = {0,1};
          3 : m[b] = y;
post  = pos(m{1}) && ={b}
```

Notice the introduction of existential quantifiers due to the use of `sp`. The invocation of the forward `rnd` tactic `rnd{1}>>;rnd{2}>>;rnd>>` to the last goal returns the goal below:

```
pre   = ={b} && (x{2} <= z{2} && z{2} <= y{2}) &&
                 (x{1} <= z{1} && z{1} <= y{1}) &&
                  (exists (x : int),
                     y{2} = x + k2 &&
                      x{2} = x + k1 &&
                       (exists (x_0 : int),
                          y{1} = x_0 + k2 &&
                           x{1} = x_0 + k1 && 0 <= x_0 && pos(m{1})))
stmt1 =   1 : m[b] = y;
stmt2 =   1 : m[b] = y;
post  = pos(m{1}) && ={b}
```

which can be further processed by `sp` returning following goal, dischargeable by `trivial`:

```
pre   = exists (l : (bool,int)map),
          l[b{2} <- y{2}] = m{2} &&
           (exists (l_0 : (bool,int)map),
```

```
          l_0[b{1} <- y{1}] = m{1} &&
           ={b} && (x{2} <= z{2} && z{2} <= y{2}) &&
                   (x{1} <= z{1} && z{1} <= y{1}) &&
                    (exists (x : int),
                       y{2} = x + k2 &&
                        x{2} = x + k1 &&
                         (exists (x_0 : int),
                            y{1} = x_0 + k2 &&
                             x{1} = x_0 + k1 && 0 <= x_0 && pos(l_0))))
stmt1 =
stmt2 =
post  = pos(m{1}) && ={b}
```

2.4.4.3 The `simpl` tactic

Computes the weakest precondition of the deterministic, loop-free suffix of the games in the judgment. It then simplifies the resulting post-condition by eliminating absurd and trivial cases. If the resulting post-condition is `true` and the resulting statement are lossless (terminate absolutely) then it resolve the goal.

2.4.4.4 The `trivial` tactic

Combines wp and rnd to simplify the goal. It tries to match random assignments in both programs applying the two-sided rule; if this fails it will apply the one-side rule. If the resulting goal contains two empty statement it try to prove the post-condition using the pre-condition. As for the tactic `simpl` 2.4.4.3 the part of the post-condition which is proved are removed. If the resulting post-condition become simply `true`, the goal is resolved.

2.4.4.5 The `auto` tactic

Computes the weakest precondition of the deterministic, loop-free suffix of the games in the judgment. When encountering procedure calls, it looks for a matching proven 'equiv' statement; if none is found it tries to prove one using the optional "rel-exp" argument as invariant. It stops when it encounters a random assignment.

2.4.4.6 The `derandomize` tactic

2.4.4.7 The `eqobs_in` tactic

Syntax `eqobs_in` (*g_eqs*) (*invariant*) (*eqs*)
where *equalities* are conjonction of equalities between variables of each side (i.e. of the form `x{1} = y{2}` or `={u,v}`)

Description The `eqobs_in` tactic applies a fast but incomplete strategy to verify goals with a particular pattern:

$$\frac{\begin{array}{cc}\models c_1' \sim c_2' : \Psi \Longrightarrow \varphi \land =\!\{Y\} & \models c_1 \sim c_2 : \varphi \land =\!\{Y\} \Longrightarrow \varphi \land =\!\{X\} \\ \vdash \varphi \land =\!\{X\} \Rightarrow \Phi & c_1, c_2 \text{ do not modify } \varphi\end{array}}{\models c_1'; c_1 \sim c_2'; c_2 : \Psi \Longrightarrow \Phi}$$

where $=\!\{X\}$ stands for the left-right equality of a set of variables X. In fact, `eqobs_in` returns only the first subgoal $\models c_1' \sim c_2' : \Psi \Longrightarrow \varphi \land =\!\{Y\}$ and computes the set Y such that the second subgoal $\models c_1 \sim c_2 : \varphi \land =\!\{Y\} \Longrightarrow \varphi \land =\!\{X\}$ holds trivially. The strategy implemented by `eqobs_in` consumes the statements from bottom to top until it fails to proceed, for instance when it finds an assignment to a variable in φ.

- The argument *invariant* is a relational formula that cannot be modified by the statements (it corresponds to φ in the rule above).
- The argument *eqs* is a relational formula, defined as a conjunction of equalities between variables of each side (i.e. of the form `x{1} = y{2}` or `={u,v}`) (it corresponds to $=\!\{X\}$ in the rule above).
- Similarly to *eqs*, the argument *g_eqs* is a conjunction of variable equalities, restricted to global variables, and required to hold as invariant of every call in the current goal.

The conjunction of *invariant* and *eqs* is required to imply the postcondition of the current goal.

Example The following example shows the result of applying the tactic
`eqobs_in (true) (0<=z{1}) (={y,l} && x{1}=w{2})` to the goal on the left:

```
pre   = ={y} && x{1} = w{2}
stmt1 =   1 : z = 1;
          2 : b = {0,1};
          3 : if (b) x = y + z;
          4 : while (x <= 10) x = x + 2;
          5 : l = 2;
stmt2 =   1 : z = 1;
          2 : b = {0,1};
          3 : if (b) w = y + z;
          4 : while (w <= 10) {
                w = w + 2;
                b = !b;
              }
          5 : l = 2;
post  = x{1} = w{2} && ={l}
```

```
pre   = ={y} && x{1} = w{2}
stmt1 =   1 : z = 1;
stmt2 =   1 : z = 1;
post  = (={z,y} && x{1} = w{2}) && 0 <= z{1}
```

Notice that:

- the computation stopped until the assignments to `z` at position 1, since it was modifying a variable occurring in the invariant `0<=z{1}`
- the conjunction of the equalities `={y,l} && x{1}=w{2}` and the invariant `0<=z{1}` implies the postcondition `x{1} = w{2} && ={l}`
- in order to establish the equality `x{1}=w{2}`, the tactic requires the equalities `x{1}=w{2}`, and `={y,z}` after the assignment to `z`. More precisely, before the while loop it is enough to require the equality `x{1}=w{2}`, but before the conditional statements both `x{1}=w{2}` and `={y,z}` are required, plus the equality on the guards (`={b}`) which is later removed by the two-sided random assignment rule.
- replacing the redundant invariant `0<=z{1}` by `true` returns a subgoal with empty statements that does not require `={z}` in the postcondition:

```
pre   = ={y} && x{1} = w{2}
stmt1 =
stmt2 =
post  = ={y} && x{1} = w{2}
```

2.4.5 by auto, by eager

- `auto [<rel-exp>]` Computes the weakest precondition of the deterministic, loop-free suffix of the games in the judgment. When encountering procedure calls, it looks for a matching proven 'equiv' statement; if none is found it tries to prove one using the optional "rel-exp" argument as invariant. It stops when it encounters a random assignment.

2.4.6 Open equiv goal

```
equiv name : G1.f1 ~ G2.f2 : pre ==> post
equiv name : G1.f1 ~ G2.f2 : (inv)
```

2.5 Miscellaneous tool directives

- `include` *filename*: Loads and processes the contents of the EasyCrypt file *filename*.
- `timeout` *secs*: Sets the current timeout given to SMT solvers to the value *secs*. Used to increase the default timeout value when no SMT solver manage to prove the required logical goals.
- `prover` *prover*$_1$,..,*prover*$_k$: Sets the list of provers (separated by '`,`') that are available to discharge the logical verification conditions. By default, EasyCrypt tries with all provers recognized when invoking `why3config --detect`. A prover name can be given either as an identifier or a string.
- `check` *name*/ `print` *name* Show information about the object associated to the name *name*.

- `checkproof`: Enables and disables the verification of logical verification conditions.
- `set` *name*/`unset` *name*: Make the axiom or lemma with name *name* available/unavailable as hypothesis for the verification of logical formulae.
- `transparent` *name*/ `opaque` *name*: Set the definition of the predicate with name *name* as transparent or opaque. If a predicate is opaque then its definition is not unfolded during the verification of logical formulae.

Chapter 3

Probability Claims and Computation

Security properties are expressed in terms of probability of events, rather than as pRHL judgments. Pleasingly, one can derive inequalities (resp. equality) about probability quantities from valid judgments. In particular, assume that the postcondition Φ implies $A\langle 1\rangle \Rightarrow B\langle 2\rangle$. Then for any programs c_1, c_2 and precondition Ψ such that $\models c_1 \sim c_2 : \Psi \Longrightarrow \Phi$ is valid and for any initial memories m_1, m_2 satisfying the precondition Ψ, we have

$$\Pr\left[c_1, m_1 : A\right] \leq \Pr\left[c_2, m_2 : B\right]$$

Up to now, EasyCrypt assume that the two games start in the same initial memory (i.e. $m_1 = m_2$), thus the equality of initial memories should imply the validity of the precondition.

3.1 Claims that follow from relational equivalences

The natural way to obtain new claims is to deduce it from a pRHL judgment. Assume we have proved a pRHL judgment of the form:

```
equiv Fact1 : Game1.Main ~ Game2.Main : true ==> ={res}.
```

Then we can deduce:

```
claim c1 : Game1.Main[res] = Game2.Main[res] using Fact1.
```

EasyCrypt will check that the equality of the initial memories implies the validity of the precondition (here `true`) and that the postcondition implies the logical equivalence of the two events (here `={res} => (res{1} <=> res{2})`).

pRHL judgments also allow proving inequality relations between probability expressions. Assume we have proved a pRHL judgment of the form:

```
equiv Fact2 : Game1.Main ~ Game2.Main :
    true ==> ={res} && (bad{1} => bad{2}).
```

Then we can deduce:

```
claim c2 : Game1.Main[res] = Game2.Main[res] using Fact2.
```

but also:

```
claim c3 : Game1.Main[res && bad] <= Game2.Main[bad] using Fact2.
```

For the last claim, EasyCrypt checks that the postcondition of the pRHL judgment (`={res} && (bad{1} => bad{2})`) and the event associated to the first game (`res{1} && bad{1}`) imply the event associated to the second game (`bad{2}`).

There is a third kind of claim which can be deduced from a pRHL judgment. This kind of judgment is closely related to the fundamental lemma (also named difference lemma).

Fundamental lemma *Let F_1 and F_2 be to distribution, and A_1, A_2, B_1, B_2 some events. Assume that*

- $\Pr\left[F_1 : B_1\right] = \Pr\left[F_2 : B_2\right]$
- $\Pr\left[F_1 : A_1 \land \neg B_1\right] = \Pr\left[F_2 : A_2 \land \neg B_2\right]$

then we have

$$|\Pr[F_1 : B_1] - \Pr[F_2 : B_2]| \leq \Pr[F_i : B_i]$$

Now assume we have proved a specification of the form:

```
equiv Fact3 : Game1.Main ~ Game2.Main :
    true ==> B1{1} <=> B2{2} && (!B1{1} => A1{1} <=> A2{2}).
```

Then we can derive the following claims:

```
claim c4_1 : Game1.Main[B1] = Game2.Main[B2]
using Fact3.
claim c4_2 : Game1.Main[!B1 && A1] = Game2.Main[!B2 && A2]
using Fact3.
```

So the two hypotheses of the fundamental lemma are satisfied. EasyCrypt allows deriving directly the conclusion of the fundamental lemma from `Fact3`:

```
claim c4 : |Game1.Main[A1] - Game2.Main[A2] | <= Game2.Main[B2]
using Fact3.
```

For this kind of claim, EasyCrypt checks that the postcondition of the pRHL judgment implies the equivalence of the bad events (here `B1` and `B2`) in the two games. Furthermore if the postcondition is valid and the bad event (here `B2`) is not set then the two events (here `A1{1}` and `A2{2}`) should be equivalent.

3.2 Claim using same and split

There is some particular case of claim which can be deduced automatically without using pRHL judgments. More precisely, the judgment $\models c \sim c : = \Longrightarrow =$ is always valid (where $=$ means the equality of the memories). Thus, we can derive some simple properties from it.

```
claim c_1 : G1.Main[res && (b || !b)] = G1.Main[res]
same.
claim c_2 : G1.Main[res && b ] <= G1.Main[res]
same.
```

Claim defined using `same` argument should relates the probability of two events A_1 and A_2 in the same game. If the comparison operator is the equality then we should have $A_1 \Leftrightarrow A_2$ (as in the claim `c_1`). If the comparison operator is the less or equal operator then we should have $A_1 \Rightarrow A_2$ (as in the claim `c_2`).

Another way to simply derive claim is to use the `split` argument.

```
claim c_3 : G1.Main[res] = G1.Main[res && bad] + G1.Main[res && !bad]
split.
```

If the comparison operator is the equality the claim should match the generic form `G.F[A] <= G.F[A&&B] + G.F[A&&!B]`. If the comparison operator is the less or equal operator then the claim should have the generic form `G.F[A] <= G.F[B] + G.F[C]`. Furthermore EasyCrypt check that $A \Rightarrow (B \vee C)$.

An exemple of use of the `split` and `same` is the proof of the fundamental lemma, assume we have proved the specification:

```
equiv Fact3 : Game1.Main ~ Game2.Main :
    true ==> B1{1} <=> B2{2} && (!B1{1} => A1{1} <=> A2{2}).
```

Then we can derive the following claims:

```
claim c4_1 : Game1.Main[B1] = Game2.Main[B2]
using Fact3.
claim c4_2 : Game1.Main[!B1 && A1] = Game2.Main[!B2 && A2]
using Fact3.
```

but also:

```
claim c4_split1 : Game1.Main[A1] = Game1.Main[B1 && A1] + Game1.Main[!B1 && A1]
split.
claim c4_split2 : Game2.Main[A2] = Game2.Main[B2 && A2] + Game2.Main[!B2 && A2]
split.
claim c4_same1 : Game1.Main[B1 && A1] <= Game1.Main[A1]
same.
claim c4_same2 : Game2.Main[B2 && A2] <= Game1.Main[A2]
same.
```

Using the claims `c4_1`, `c4_2`, `c4_split1`, `c4_split2`, `c4_same1`, `c4_same2` the automatic provers (like `alt-ergo`) are able to derive the following claim:

```
claim c4 : |Game1.Main[A1] - Game2.Main[A2] | <= Game2.Main[B2].
```

3.3 Deducing claim from other claims

Claim can be derived as a consequence of other claims. When no argument is given after the statement of the claim EasyCrypt try to prove it using the previously proved claims.

Assume we have already proved the following claims:

```
claim c_1 : G1.Main[res] = G2.Main[res].
claim c_2 : | G2.Main[res] - G3.Main[res] | <= G3.Main[bad].
claim c_3 : G3.Main[res] = 1%r/2%r.
claim c_4 : G3.Main[bad] <= 1%r/(2^n)%r.
```

Then the following claim is automatically deduced from the previous one:

```
claim c_5 : | G1.Main[res] - 1%r/2%r | <= 1%r/(2^n)%r.
```

3.4 Claims by direct computation

During a reduction proof, we sometime need to compute or to bound the probability of an event in a given game. This can be done using the `compute` argument. Assume we have the following game:

```
game G = {
   ...
   fun Main() : bool = {
     (pk,sk) = KG();
     (m0,m1) = A_1(pk);
     c       = {0,1}^k;
     b'      = A_2(c);
     b       = {0,1};
     return b = b';
  }
}
```

Then EasyCrypt is able to compute the probability of `res=true` in the function `G.Main`:

```
claim c : G.Main[res] = 1%r/2%r
compute.
```

The `compute` argument is also able to prove the claim that can be derive using `split` and `same`, but it is less efficient. On the other side it is also more powerful, for example we can prove:

```
claim c : G.Main[A || B || C] <= G.Main[A] + G.Main[B] + G.Main[C]
compute.
```

This claim can also be obtained using the `split` argument, using the following sequence:

```
claim c_1 :  G.Main[A || B || C] <= G.Main[A || B] + G.Main[C]
split.
claim c_2 :  G.Main[A || B] <= G.Main[A] + G.Main[B]
split.
claim c : G.Main[A || B || C] <= G.Main[A] + G.Main[B] + G.Main[C].
```

The claim `c` is a direct consequence of the claims `c_1` and `c_2`.

A last example of use for `compute` is the following, assume we have a game of the form:

```
game G = {
   ...
   fun Main () : bool = {
     x = init();
     d = A(x);
     z = {0,1}^k;
     return d;
   }
}
```

Then `compute` is able to prove the following claim:

```
claim c :
  G.Main[res && mem(z,L) && length(L) <= q] <= q%r/(2^k)%r * G.Main[res]
compute.
```

3.5 Using the Failure Event Lemma to prove claims

It is often the case that the event whose probability one wants to bound is a *failure* event that can only be triggered during an oracle call. Assume that this oracle can be called at most, say q, times, and that one knows an upper bound u for the probability that failure is triggered during a single call. One can then conclude that the probability that failure is triggered during the game is bounded by $q \cdot u$. A generalization of this kind of reasoning is automated as an extension to the `compute` mechanism. Consider for instance the following game:

```
game G = {
  var n : int
  var bad : bool

  fun O(x:bitstring{k}) : bitstring{k} = {
    var y : bitstring{k} = {0,1}^k;
    if (n < q) {
      n = n + 1;
      if (x = y) bad = true;
    }
    return y;
  }

  abs A = A {O}

  fun Main() : unit = {
    n = 0;
    bad = false;
    A();
  }
}
```

The probability that `bad` is set during a single call to `O` is exactly $1/2^k$, and the number of calls made so far is given by the value of the counter `n`. Thus, the probability that failure occurs, indicated by the `bad` boolean flag, assuming at most `q` calls are made, can be bounded as follows:

```
claim pr_bad : G.Main[bad && n <= q] <= q%r * (1%r / (2^k)%r)
compute 2 bad, n.
```

The second argument `bad` to `compute` is a boolean expression that indicates the failure event, the third argument `n` is an integer expression that acts as the counter. The first argument `2` indicates the number of instructions in a prefix of the `Main` procedure; after this prefix is executed the value of the counter must be set to `0`, and the boolean expression indicating failure must be set to `false`. EasyCrypt then checks that any call to a procedure that may trigger failure either strictly increases the value of the counter, or does not decrease it but neither triggers failure. Moreover, whether the counter is increased or not must be fully determined upon entering the procedure and must not depend on its internal choices. Finally, EasyCrypt tries to compute an upper bound `u` of the probability that a single call to any procedure in the game triggers failure. If it succeeds, and the event considered is of the form `bad && n <= q`, then its probability is proven to be bounded by `q%r * u`. One may typically then prove that the bound `n <= q` is actually enforced by the game, and so the bound also holds for `bad` alone.

3.6 Claims by auto

One may automatically prove claims that follow from an equivalence that can be proved using `equiv ... by auto`:

```
claim G1_G2 : G1.Main[e1] <= G2.Main[e2]
auto.
```

is a shortcut for:

```
equiv G1_G2 : G1.Main ~ G2.Main : true ==> (e1{1} => e2{2}) by auto.
claim G1_G2 : G1.Main[e1] <= G2.Main[e2] using G1_G2.
```

3.7 Admitting claims

Claims may be also admitted without proof:

```
claim c : G.Main[res] = G'.Main[res]
admit.
```

However, be aware that admitting invalid claims can lead to inconsistencies.

Chapter 4

Example: elgamal

(The syntax used in this section may be outdated.)

We illustrate the key ingredients presented in prevous chapters with a simple example: a game-based proof of the IND-CPA-security of the ElGamal public-key encryption scheme.

The ElGamal encryption scheme is based on any cyclic group G of order q with generator g and is defined by the following triple of algorithms

- The key generation algorithm $\mathcal{KG}()$ selects uniformly a random number x from $\{0, \ldots, q-1\}$; the secret (private) key is x, the public key is g^x.
- Given a public key pk and a plaintext m (an element of the group G), the encryption algorithm $\mathcal{E}(pk, m)$ chooses uniformly a random element y from $\{0, \ldots, q-1\}$ and returns the ciphertext $(g^y, pk^y * m)$.
- Given a secret key sk and a ciphertext c, the decryption algorithm $\mathcal{D}(sk, c)$, parses c as (β, ζ) and returns a plaintext computed as $\zeta * \beta^{-x}$.

We start by declaring a type for elements of the group G, and defining type synonyms for the type of public and secret keys, plaintexts and ciphertexts:

```
type group
type skey = int
type pkey = group
type plaintext = group
type ciphertext = group * group
```

The order of the group q and its generator g are declared as constants:

```
cnst q : int
cnst g : group
```

We then declare operators that will denote the group law in G, exponentiation and discrete logarithm (in base g).

```
op (*) : group, group -> group = group_mult
op (^) : group, int -> group   = group_pow
op log : group-> int           = group_log
```

At this point the operators and constants that we declared above are completely abstract, nothing is known about them besides their type. To specify

At that point nothing say that the type `group` is a cyclic group, we only known that the type come with three operators `*`, `^` and `log`. We should specify the behavior of the operators this is done using axioms:

```
axiom q_pos : {0 < q}

axiom group_pow_add :
 forall (x:int, y:int). { g ^ (x + y) == g ^ x * g ^ y }

axiom group_pow_mult :
```

```
 forall (x:int, y:int). { (g ^ x) ^ y == g ^ (x * y) }

axiom log_pow :
 forall (g':group). { g ^ log(g') == g' }

axiom pow_mod :
 forall (z:int). { g ^ (z%q) == g ^ z }
```

The first axiom `q_pos` expresses that the integer `q` representing the order of the group is positive. The next `group_pow_add` and `group_pow_mult` specify the behavior of the multiplication and the exponentiation, `log_pow` partially specify the behavior of the logarithm operator. The `+` operator used in `group_pow_add` is the predefined additive operator over integer. Note that the `*` operator in the axiom `group_pow_mult` represent the multiplication over integer and not the multiplication law of the group (EasyCrypt allows to overloading of operator). The last axiom expresses the fact that the group is a cyclic group of order `q`, `%` stand for the modulus operator over integer.

To be able to perform the proof we also add axioms on the modulus operator:

```
axiom mod_add :
 forall (x:int, y:int). { (x%q + y)%q == (x + y)%q }

axiom mod_small :
 forall (x:int). { 0 <= x } => { x < q } => { x%q == x}

axiom mod_sub :
 forall (x:int, y:int). { (x%q - y)%q == (x - y)%q }
```

The IND-CPA semantic security is expressed as a game parameterized by an pair of adversaries, let us declare this two adversaries:

```
adversary A1(pk:pkey)              : plaintext * plaintext {}
adversary A2(pk:pkey, c:ciphertext) : bool {}
```

The first one `A1` expect a public key `pk` and return a pair of plaintext, the second one expect a public key and a cyphertext and return a boolean. The semi-bracket contains the declaration of the oracles that can be used by the adversaries, here there is no oracles.

We can now define the game representing the IND-CPA semantic security of ElGamal:

```
game INDCPA = {
  fun KG() : keys = {
    var x : int = [0..q-1];
    return (x, g^x);
  }

  fun Enc(pk:pkey, m:plaintext): ciphertext = {
    var y : int = [0..q-1];
    return (g^y, (pk^y) * m);
  }

  abs A1 = A1 {}
  abs A2 = A2 {}

  fun Main() : bool = {
    var sk : skey;
    var pk : pkey;
    var m0, m1, mb : plaintext;
    var c: ciphertext;
    var b, b' : bool;

    (sk,pk) = KG();
    (m0,m1) = A1(pk);
    b = {0,1};
    mb = b ? m0 : m1;
    c = Enc(pk, mb);
```

```
    b' = A2(pk, c);
    return (b == b');
  }
}
```

The game start by the declaration of two functions the key generation algorithm `KG` and the encryption algorithm `Enc`. Then come the definition of the two adversary `A1` and `A2`, they are defined to be equal to the abstract functions previously defined. The main function, at the end of the game, represent the IND-CPA experiment. First the key generation algorithm is used to generate the secret and public keys, then the public key is given to `A1` which generate two plaintext `m0` and `m1`. The instruction `b = {0,1}` uniformly sample a boolean which is stored in `b`. Depending on this bit `b` either the plaintext `m0` or `m1` is encrypted with the public key `pk`, generating the ciphertext `c`. The public key and ciphertext are then give back to the adversary `A2`. The goal of the adversary is to discover which plaintext as been encrypted. It win if `b` is equal to `b'`.

The IND-CPA semantic security of ElGamal express that there exists a adversary `B` build on top of `A1` and `A2` which as a higher probability of breaking the Decisional Diffie Hellman problem (DDH) than `A1` and `A2` of winning the IND-CPA game. The first thing to do is to define the two games and the adversary `B` involved in DDH problem:

```
game DDH0 = {
  abs A1 = A1 {}
  abs A2 = A2 {}

  fun B(gx:group, gy:group, gz:group) : bool = {
    var m0, m1, mb : plaintext;
    var c : ciphertext;
    var b, b' : bool;

    (m0, m1) = A1(gx);
    b = {0,1};
    mb = b ? m0 : m1;
    c = (gy, gz * mb);
    b' = A2(gx,c);
    return (b == b');
  }

  fun Main() : bool = {
    var x, y : int;
    var d : bool;

    x = [0..q-1];
    y = [0..q-1];
    d = B(g^x, g^y, g^(x*y));
    return d;
  }
}

game DDH1 = DDH0 where
  Main = {
    var x, y, z : int;
    var d : bool;

    x = [0..q-1];
    y = [0..q-1];
    z = [0..q-1];
    d = B(g^x, g^y, g^z);
    return d;
  }
```

The main experiment in the game `DDH0` start by uniformly sample two values `x` and `y` between 0 and $q-1$ and then send g^x, g^y, g^{xy} to the adversary `B`. The game `DDH1` is defined to be equal to the game

DDH0 where only the main function changes: a new variable z is uniformly sample and g^z is send to the adversary instead of g^{xy}. The goal of the adversary is to discover if its last argument correspond to g^{xy} or g^z, i.e. if it play between DDH0 or DDH1.

We can know start our proof:

```
prover alt-ergo

equiv auto Fact1 : INDCPA.Main ~ DDH0.Main : {true} ==> ={res};;

claim Pr1 : INDCPA.Main[res] == DDH0.Main[res]
using Fact1;;
```

The first line select the prover to be used, here `alt-ergo` (the default one is `simplify`). The second line is the main component of EasyCrypt. We demonstrate using the probabilistic Relational Hoare Logic (pRHL) that the two functions `INDCPA.Main` and `DDH0.Main` are indistinguishable if we observe only their results. This allows to proving the claim `Pr1` which state that the probability that `res` is true after running the two programs is equal.

```
game G1 = INDCPA where
  Main = {
    var x, y, z : int;
    var gx, gy, gz : group;
    var d, b, b' : bool;
    var m0, m1, mb : plaintext;
    var c : ciphertext;

    x = [0..(q - 1)];
    y = [0..(q - 1)];
    gx = g^x;
    gy = g^y;
    (m0, m1) = A1 (gx);
    b = {0,1};
    mb = b ? m0 : m1;
    z = [0..(q - 1)];
    gz = g^z;
    c = (gy, gz * mb);
    b' = A2 (gx, c);
    d = (b == b');
    return d;
  }

equiv auto Fact2 : G1.Main ~ DDH1.Main : {true} ==> ={res};;

claim Pr2 : G1.Main[res] == DDH1.Main[res]
using Fact2;;

game G2 = G1 where
  Main = {
    var x, y, z : int;
    var gx, gy, gz : group;
    var d, b, b' : bool;
    var m0, m1, mb : plaintext;
    var c : ciphertext;

    x = [0..(q - 1)];
    y = [0..(q - 1)];
    gx = g^x;
    gy = g^y;
    (m0, m1) = A1(gx);
    z = [0..(q - 1)];
    gz = g^z;
    c = (gy, gz);
```

```
    b’ = A2 (gx, c);
    b = {0,1};
    d = (b == b’);
    return d;
  }

equiv Fact3 : G1.Main ~ G2.Main : {true} ==> ={res}
 swap{2} [10-10] -4; auto;
 rnd (z + log(b?m0:m1)) % q, (z - log(b?m0:m1)) % q; wp; rnd;
 auto; repeat rnd;
 trivial;;
save;;

claim Pr3 : G1.Main[res] == G2.Main[res]
using Fact3;;

claim Pr4 : G2.Main[res] == 1%r / 2%r
compute;;

claim Conclusion :
 | INDCPA.Main[res] - 1%r / 2%r | <= | DDH0.Main[res] - DDH1.Main[res] |
```

Part II

Language Reference

4.1 Lexical conventions

Comments.

Comments are enclosed by (∗ and ∗).

Strings.

Identifiers.

$\langle \mathit{letter} \rangle$ ::= ‘a’ - ‘z’ | ‘A’ - ‘Z’ | ‘_’

$\langle \mathit{digit} \rangle$ ::= ‘0’ - ‘9’

$\langle \mathit{other\_letter} \rangle$ ::= $\langle \mathit{letter} \rangle$ | $\langle \mathit{digit} \rangle$ | ‘’’

$\langle \mathit{ident} \rangle$ ::= $\langle \mathit{letter} \rangle$ $\langle \mathit{other\_letter} \rangle^*$

$\langle \mathit{ident\_list} \rangle$ ::= $\langle \mathit{ident} \rangle$ | $\langle \mathit{ident} \rangle$ ‘,’ $\langle \mathit{ident\_list} \rangle$

$\langle \mathit{ident\_list0} \rangle$ ::= $\langle \mathit{empty} \rangle$ | $\langle \mathit{ident\_list} \rangle$

$\langle \mathit{prim\_ident} \rangle$ ::= ‘’’ $\langle \mathit{ident} \rangle$

$\langle \mathit{prim\_ident\_list} \rangle$ ::= $\langle \mathit{prim\_ident} \rangle$ | $\langle \mathit{prim\_ident\_list} \rangle$ ‘,’ $\langle \mathit{prim\_ident\_list} \rangle$

$\langle \mathit{number\_list} \rangle$ ::= $\langle \mathit{number} \rangle$ | $\langle \mathit{number} \rangle$ ‘,’ $\langle \mathit{number\_list} \rangle$

$\langle \mathit{qualif\_fct\_name} \rangle$ ::= $\langle \mathit{ident} \rangle$‘.’$\langle \mathit{ident} \rangle$

$\langle \mathit{number} \rangle$ ::= $\langle \mathit{digit} \rangle^+$

$\langle \mathit{znumber} \rangle$ ::= $\langle \mathit{number} \rangle$ | ‘-’$\langle \mathit{number} \rangle$

Keywords.

The following literals are reserved and must not be used as identifiers:

Operators.

$\langle \mathit{op\_char} \rangle$ ::= ‘=’ | ‘<’ | ‘>’ | ‘~’ | ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’
| ‘!’ | ‘$’ | ‘&’ | ‘?’ | ‘@’ | ‘^’ | ‘.’ | ‘:’ | ‘|’ | ‘#’

$\langle \mathit{bin\_op} \rangle$ ::= $\langle \mathit{op\_char} \rangle^+$

$\langle \mathit{u\_op} \rangle$ :: = ‘-’ | ‘!’

$\langle \mathit{op\_ident} \rangle$ ::= $\langle \mathit{ident} \rangle$ | ‘(’ $\langle \mathit{bin\_op} \rangle^+$ ‘)’

4.2 Type Expressions.

$\langle \mathit{type} \rangle$ ::= $\langle \mathit{ident} \rangle$
| ’ $\langle \mathit{ident} \rangle$
| $\langle \mathit{type} \rangle$ $\langle \mathit{ident} \rangle$
| ($\langle \mathit{type} \rangle$ (‘,’ $\langle \mathit{type} \rangle)^+$) $\langle \mathit{ident} \rangle$

```
                 |  ( ⟨type⟩ (‘*’ ⟨type⟩)+ )
                 |  ‘bitstring’ ‘{’ ⟨type⟩ ‘}’
                 |  ‘(’ ⟨type⟩ ‘)’

⟨typed_vars⟩    ::= ⟨ident_list⟩ ‘:’ ⟨type⟩

⟨typed_var_list⟩ ::= ⟨typed_vars⟩ | ⟨typed_vars⟩ ‘,’ ⟨typed_var_list⟩

⟨param_list⟩    ::= ⟨empty⟩ | ⟨typed_var_list⟩

⟨param_decl⟩    ::= ‘(’ ⟨param_list⟩ ‘)’

⟨type_list⟩     ::= ⟨type⟩ ‘,’ ⟨type⟩
                 |  ⟨type⟩ ‘,’ ⟨type_list⟩

⟨type_list0⟩    ::= ⟨type⟩
                 |  ‘(’ ⟨type_list⟩ ‘)’
                 |  ‘()’

⟨fun_type⟩      ::= ⟨type_list0⟩ ‘->’ ⟨type⟩

⟨fun_type_list⟩ ::= ⟨fun_type⟩ | ⟨fun_type⟩ ‘;’ ⟨fun_type_list⟩

⟨fun_type_list0⟩ ::= ⟨empty⟩ | ⟨fun_type_list⟩
```

4.3 Expressions.

Simple expressions:

```
⟨simpl_exp⟩     ::= ⟨number⟩
                 |  ⟨ident⟩
                 |  ⟨simpl_exp⟩ ‘[’ ⟨exp⟩ ‘]’
                 |  ⟨simpl_exp⟩ ‘[’ ⟨exp⟩ ‘<-’ ⟨exp⟩ ‘]’
                 |  ⟨ident⟩ ‘(’ ⟨exp_list0⟩ ‘)’
                 |  ⟨simpl_exp⟩ ‘{’ ‘{’⟨number⟩‘}’ ‘}’
                 |  ⟨simpl_exp⟩ ‘%r’
                 |  ⟨qualif_fct_name⟩ ‘[’ ⟨exp⟩ ‘]’
                 |  ‘(’ ⟨exp⟩ ‘,’ ⟨exp_list⟩ ‘)’
                 |  ‘(’ ⟨exp⟩ ‘)’
                 |  ‘[’ ⟨exp_list⟩ ‘]’
                 |  ‘=’ ‘{’ ⟨pos_ident_list⟩ ‘}’
                 |  ‘|’ ⟨exp⟩ ‘|’
                 |  ⟨simpl_exp⟩ ‘{’⟨number⟩‘}’
```

Random expressions:

```
⟨rnd_exp⟩       ::= ‘{’ ⟨number⟩ ‘,’ ⟨number⟩ ‘}’
                 |  ‘{’ ⟨number⟩ ‘,’ ⟨number⟩ ‘}^’ ⟨exp⟩
                 |  ‘[’ ⟨exp⟩ ‘..’ ⟨exp⟩ ‘]’
                 |  ‘(’⟨rnd_exp⟩ ‘\’ ⟨exp⟩ ‘)’
```

General expressions:

```
⟨exp⟩          ::= ⟨exp⟩ ⟨bin_op⟩ ⟨exp⟩
                 |  ⟨u_op⟩ ⟨exp⟩
                 |  ⟨exp⟩ '?' ⟨exp⟩ ':' ⟨exp⟩
                 |  'if' ⟨exp⟩ 'then' ⟨exp⟩ 'else' ⟨exp⟩
                 |  'forall' ⟨param_decl⟩ ['['⟨trigger_list⟩']'] ',' ⟨exp⟩
                 |  'exists' ⟨param_decl⟩ ['['⟨trigger_list⟩']'] ',' ⟨exp⟩
                 |  'let' ⟨ident_list⟩ '=' ⟨exp⟩ 'in' ⟨exp⟩
                 |  ⟨simpl_exp⟩
                 |  ⟨rnd_exp⟩

⟨trigger_list⟩ ::= ⟨trigger⟩ | ⟨trigger⟩ '|' ⟨trigger_list⟩

⟨trigger⟩      ::= ⟨exp⟩ | ⟨exp⟩ ',' ⟨trigger⟩
```

4.4 Declarations.

type

```
⟨poly_type⟩ ::= '(' ⟨prim_ident_list⟩ ')' | ⟨prim_ident⟩

⟨type_elem⟩ ::= 'type' [⟨poly_type⟩] ⟨ident⟩
              |  'type' [⟨poly_type⟩] ⟨ident⟩ '=' ⟨type⟩
```

cnst

```
⟨cnst_elem⟩ ::= 'cnst' ⟨ident_list⟩ ':' ⟨type⟩
              |  'cnst' ⟨ident_list⟩ ':' ⟨type⟩ '=' ⟨exp⟩
```

op

```
⟨op_body⟩ ::= ':' ⟨fun_type⟩
            |  ⟨param_decl⟩ '=' ⟨exp⟩

⟨op_elem⟩ ::= 'op' ⟨op_ident⟩ ⟨op_body⟩
            |  'op' ⟨op_ident⟩ ⟨op_body⟩ 'as' ⟨ident⟩
```

pop

```
⟨pop_elem⟩ ::= 'pop' ⟨op_ident⟩ ':' ⟨fun_type⟩
```

pred

```
⟨pred_elem⟩ ::= 'pred' ⟨ident⟩ ⟨param_decl⟩ '=' ⟨exp⟩
              |  'pred' ⟨ident⟩ ':' ⟨type_ist⟩
```

axiom

```
⟨axiom_elem⟩ ::= 'axiom' ⟨ident⟩ ':' ⟨exp⟩
              |  'lemma' ⟨ident⟩ ':' ⟨exp⟩
```

adversary

```
⟨adv_elem⟩ ::= 'adversary' ⟨fun_decl⟩ '{' ⟨fun_type_list0⟩ '}'
```

Games.

```
⟨base_instr⟩   ::= ⟨ident⟩ '('⟨exp_list0⟩ ')'
                |  ⟨ident⟩ '=' ⟨exp⟩
                |  '(' ⟨ident_list⟩ ')' '=' ⟨exp⟩
                |  ⟨ident⟩ '[' ⟨exp⟩ ']' '=' ⟨exp⟩

⟨instr⟩        ::= ⟨base_instr⟩ ;
                |  'if' '(' ⟨exp⟩ ')' ⟨block⟩ 'else' ⟨block⟩
                |  'if' '(' ⟨exp⟩ ')' ⟨block⟩
                |  'while' '(' ⟨exp⟩ ')' ⟨block⟩

⟨block⟩        ::= ⟨base_instr⟩ ';'
                |  '{' ⟨stmt⟩ '}'

⟨stmt⟩         ::= ⟨instr⟩ ⟨stmt⟩
                |  ⟨empty⟩

⟨ret_stmt⟩     ::= 'return' ⟨exp⟩ ';'

⟨loc_decl⟩     ::= 'var' ⟨ident_list⟩ ':' ⟨type⟩ ['=' ⟨exp⟩ ]';'

⟨loc_decl_list⟩ ::= ⟨loc_decl⟩+

⟨fun_def_body⟩ ::= '{' [⟨loc_decl_list⟩] ⟨stmt⟩ [⟨ret_stmt⟩] '}'

⟨fun_decl⟩     ::= ⟨ident⟩ ⟨param_decl⟩ ':' ⟨type⟩

⟨pg_elem⟩      ::= 'var' ⟨ident_list⟩ ':' ⟨type⟩
                |  'fun' ⟨fun_decl⟩ '=' ⟨fun_def_body⟩
                |  'fun' ⟨ident⟩ '=' ⟨qualif_fct_name⟩
                |  'abs' ⟨ident⟩ '=' ⟨ident⟩ '{' ⟨ident_list0⟩ '}'

⟨game_elem⟩    ::= 'game' ⟨ident⟩ '=' '{' ⟨pg_elem⟩* '}'
                |  'game' ⟨ident⟩ '=' ⟨ident⟩ ⟨var_modifier⟩ 'where' ⟨redef_list⟩
```

4.5 pRHL judgments

equiv

```
⟨inv_info⟩     ::= '(' ⟨exp⟩ ')'
                |  'upto' '(' ⟨exp⟩ ')' ['and' '(' ⟨exp⟩ ')' ] ['with' '(' ⟨exp⟩ ')']

⟨auto_info⟩    ::= [⟨inv_info⟩] ['using' ⟨ident_list⟩]
```

⟨*equiv_concl*⟩ ::= ⟨*exp*⟩ '==>' ⟨*exp*⟩
| ⟨*exp*⟩ '=' '(' ⟨*exp*⟩ ':' ⟨*exp*⟩ ')' '=>' ⟨*exp*⟩
| ⟨*inv_info*⟩

⟨*equiv_elem*⟩ ::= 'equi' ⟨*ident*⟩ ':' ⟨*qualif_fct_name*⟩ '~' ⟨*qualif_fct_name*⟩ ':' ⟨*equiv_concl*⟩
| 'equi' ⟨*ident*⟩ ':' ⟨*qualif_fct_name*⟩ '~' ⟨*qualif_fct_name*⟩ ':' ⟨*equiv_concl*⟩ 'by' 'auto' ⟨*auto_info*⟩
| 'equi' ⟨*ident*⟩ ':' ⟨*qualif_fct_name*⟩ '~' ⟨*qualif_fct_name*⟩ ':' ⟨*equiv_concl*⟩ 'by' 'eager' ⟨*block*⟩

4.6 Tactics

⟨*interval*⟩ ::= '[' ⟨*number*⟩ '-' ⟨*number*⟩ ']' | ⟨*number*⟩

⟨*rnd_info*⟩ ::= '(' ⟨*exp*⟩ ')' ',' '(' ⟨*exp*⟩ ')' | '(' ⟨*exp*⟩ ')' | '{' ⟨*number*⟩'}'

⟨*side_at_pos*⟩ ::= ['{'⟨*number*⟩'}'] ['at' ⟨*number_list*⟩ | 'last']

⟨*inline_info*⟩ ::= 'at' ⟨*number_list*⟩ | 'last' | ⟨*ident_list*⟩

⟨*tactic*⟩ ::= 'idtac'
| 'call' ⟨*auto_info*⟩
| 'inline' ['{'⟨*number*⟩'}'] [⟨*inline_info*⟩]
| 'asgn'
| 'rnd' [⟨*rnd_info*⟩]
| 'swap' ['{'⟨*number*⟩'}'] ⟨*interval*⟩ ⟨*znumber*⟩
| 'swap' ['{'⟨*number*⟩'}'] ⟨*znumber*⟩
| 'simpl'
| 'trivial'
| 'auto' ⟨*auto_info*⟩
| 'rauto' ⟨*auto_info*⟩
| 'derandomize' ['{'⟨*number*⟩'}']
| 'wp'
| 'case' ['{'⟨*number*⟩'}'] ':' ⟨*exp*⟩
| 'if' ['{'⟨*number*⟩'}']
| 'condt' ⟨*side_at_pos*⟩
| 'condf' ⟨*side_at_pos*⟩
| 'while' ⟨*side_at_pos*⟩ ':' ⟨*exp*⟩
| 'while' ⟨*side_at_pos*⟩ ':' ⟨*exp*⟩ ':' ⟨*exp*⟩ ',' ⟨*exp*⟩
| 'while' ⟨*exp*⟩ ',' ⟨*exp*⟩ ',' ⟨*exp*⟩ ',' ⟨*exp*⟩ ',' ⟨*exp*⟩ ':' ⟨*exp*⟩
| 'apply' ⟨*ident*⟩ '(' ⟨*exp_list0*⟩ ')'
| 'pRHL'
| 'apRHL'
| 'unroll' ⟨*side_at_pos*⟩
| 'strengthen' ⟨*side_at_pos*⟩ ':' ⟨*exp*⟩
| 'app' ⟨*number*⟩ ⟨*number*⟩ ⟨*exp*⟩
| 'app' ⟨*number*⟩ ⟨*number*⟩ ⟨*exp*⟩ ':' ⟨*exp*⟩ ',' ⟨*exp*⟩ ':' ⟨*exp*⟩ ',' ⟨*exp*⟩
| 'try' ⟨*tactics_paren*⟩
| '*' ⟨*tactics_paren*⟩
| '!' ⟨*number*⟩ ⟨*tactics_paren*⟩
| 'admit'
| 'expand' ⟨*ident_list0*⟩
| 'let' ⟨*side_at_pos*⟩ ⟨*ident*⟩ ':' ⟨*type*⟩ '=' ⟨*exp*⟩

⟨*subgoal_tactics*⟩ ::= [⟨*tactics*⟩] '|' ⟨*subgoal_tactics*⟩ | [⟨*tactics*⟩]

```
⟨tactic2⟩       ::= ⟨tactic⟩ | '[' ⟨subgoal_tactics⟩ ']' | '('⟨tactics⟩')'

⟨tactic_list⟩   ::= ⟨tactic2⟩ ';' ⟨tactic_list⟩ | ⟨tactic2⟩

⟨tactics⟩       ::= ⟨tactic⟩ ';' ⟨tactic_list⟩ | ⟨tactic⟩

⟨tactics_paren⟩ ::= ⟨tactic⟩ | '(' ⟨tactics⟩ ')'
```

4.7 Probability claims

claim

```
⟨claim_elem⟩ ::= 'claim' ⟨ident⟩ ':' ⟨exp⟩
              |  'claim' ⟨ident⟩ ':' ⟨exp⟩ 'admit'
              |  'claim' ⟨ident⟩ ':' ⟨exp⟩ 'compute'
              |  'claim' ⟨ident⟩ ':' ⟨exp⟩ 'split'
              |  'claim' ⟨ident⟩ ':' ⟨exp⟩ 'same'
              |  'claim' ⟨ident⟩ ':' ⟨exp⟩ 'using' ⟨ident⟩
              |  'claim' ⟨ident⟩ ':' ⟨exp⟩ 'compute' ⟨number⟩ ⟨exp⟩ ',' ⟨exp⟩
```

4.7.1 Program

```
⟨global_elem⟩ ::= 'include' '"' ⟨string⟩ '"'
               |  ⟨type_elem⟩
               |  ⟨cnst_elem⟩
               |  ⟨op_elem⟩
               |  ⟨pop_elem⟩
               |  ⟨pred_elem⟩
               |  ⟨axiom_elem⟩
               |  ⟨adv_elem⟩
               |  ⟨game_elem⟩
               |  ⟨equiv_elem⟩
               |  ⟨claim_elem⟩
               |  ⟨tactics⟩
               |  'save'
               |  'abort'
               |  'set' ⟨ident_list⟩
               |  'unset' ⟨ident_list⟩
               |  'prover' ⟨prover_list⟩
               |  'checkproof'
               |  'transparent' ⟨ident_list⟩
               |  'opaque' ⟨ident_list⟩
               |  'timeout' ⟨number⟩
               |  'check' ⟨check⟩
               |  'print' ⟨print⟩

⟨program⟩     ::= ⟨global_elem⟩ '.'
               |  ⟨global_elem⟩ '.' ⟨program⟩
```