EasyCrypt Reference Manual

Version 1.x — Compiled on February 18, 2018

Contents

1 Getting Started **4**
- 1.1 Introduction . . . 4
- 1.2 Installing EasyCrypt . . . 4
- 1.3 Running EasyCrypt . . . 4
- 1.4 More Information . . . 6
- 1.5 Bug Reporting . . . 6
- 1.6 About this Documentation . . . 6

2 Specifications **7**
- 2.1 Lexical Categories . . . 8
- 2.2 Script Structure, Name spaces, Printing and Searching . . . 10
- 2.3 Expressions Language . . . 10
  - 2.3.1 Type Expressions . . . 10
  - 2.3.2 Type Declarations . . . 12
  - 2.3.3 Expressions and Operator Declarations . . . 13
- 2.4 Module System . . . 18
  - 2.4.1 Modules . . . 18
  - 2.4.2 Module Types . . . 21
  - 2.4.3 Global Variables . . . 26
- 2.5 Logics . . . 27
  - 2.5.1 Formulas . . . 27
  - 2.5.2 Axioms and Lemmas . . . 33

3 Tactics **35**
- 3.1 Proof Engine . . . 35
- 3.2 Matching . . . 38
- 3.3 Ambient logic . . . 39
  - 3.3.1 Proof Terms . . . 39
  - 3.3.2 Occurrence Selectors and Rewriting Directions . . . 40
  - 3.3.3 Introduction and Generalization . . . 40
  - 3.3.4 Tactics . . . 47
  - 3.3.5 Tacticals . . . 69
- 3.4 Program Logics . . . 72
  - 3.4.1 Tactics for Reasoning about Programs . . . 72
  - 3.4.2 Tactics for Transforming Programs . . . 103
  - 3.4.3 Tactics for Reasoning about Specifications . . . 130
  - 3.4.4 Automated Tactics . . . 170
  - 3.4.5 Advanced Tactics . . . 177

4 Structuring Specifications and Proofs **200**
- 4.1 Theories . . . 200
- 4.2 Sections . . . 200

5 EasyCrypt Library **201**

6 Advanced Features and Usage **202**

7 Examples **203**
7.1 Hashed ElGamal 203
7.2 BR93 203

References **204**

Index **205**

Chapter 1

Getting Started

1.1 Introduction

EasyCrypt [BDG+14, BGHZ11] is a framework for interactively finding, constructing, and machine-checking security proofs of cryptographic constructions and protocols using the code-based sequence of games approach [BR04, BR06, Sho04]. In EasyCrypt, cryptographic games and algorithms are modeled as *modules*, which consist of procedures written in a simple user-extensible imperative language featuring while loops and random sampling operations. Adversaries are modeled by *abstract* modules—modules whose code is not known and can be quantified over. Modules may be parameterized by abstract modules.

EasyCrypt has four logics: a probabilistic, relational Hoare logic (pRHL), relating pairs of procedures; a probabilistic Hoare logic (pHL) allowing one to carry out proofs about the probability of a procedure's execution resulting in a postcondition holding; an ordinary (possibilistic) Hoare logic (HL); and an ambient higher-order logic for proving general mathematical facts and connecting judgments in the other logics. Once lemmas are expressed, proofs are carried out using *tactics*, logical rules embodying general reasoning principles, and which transform the current lemma (or *goal*) into zero or more *subgoals*—sufficient conditions for the original lemma to hold. Simple ambient logic goals may be automatically proved using SMT solvers. Proofs may be structured as sequences of lemmas, and EasyCrypt's *theories* may be used to group together related types, predicates, operators, modules, axioms and lemmas. Theory parameters that may be left abstract when proving its lemmas—types, operators and predicates—may be instantiated via a *cloning* process, allowing the development of generic proofs that can later be instantiated with concrete parameters.

1.2 Installing EasyCrypt

EasyCrypt may be found on GitHub.

`https://github.com/EasyCrypt/easycrypt`

Detailed building instructions for EasyCrypt and its dependencies and supporting tools can be found in the project's README file.[1]

1.3 Running EasyCrypt

EasyCrypt scripts resides in files with the `.ec` suffix. (As we will see in Chapter 4, EasyCrypt also has *abstract* theories, which must be cloned before being used. Such theories reside in files with the `.eca` suffix.)

To run EasyCrypt in *batch* mode, simply invoke it from the shell, giving it an EasyCrypt script—with suffix `.ec`—as argument:

[1] `https://github.com/EasyCrypt/easycrypt/blob/1.0/README.md`

```
easycrypt file.ec
```

EasyCrypt will display its progress as it checks the file. Information about EasyCrypt's command-line arguments can be found in Chapter 6.

When developing EasyCrypt scripts, though, EasyCrypt can be run *interactively*, as a subprocess of the Emacs text editor. One's interaction with EasyCrypt is mediated by Proof General, a generic Emacs front-end for proof assistants. Upon visiting an EasyCrypt file, the "Proof-General" tab of the Emacs menu may be used execute the file, step-by-step, as well as to undo steps, etc. Information about the "EasyCrypt" menu tab may be found in Chapter 6.

A sample EasyCrypt script is shown in Listing 1.1.

```
(* Load (require) the theories Bool and DBool, and import their
   definitions and lemmas into the environment. Bool defines the
   exclusive-or operator (^^), and DBool defines the uniform
   distribution on the booleans ({0,1}). *)

require import Bool DBool.

(* G1.f() yields a randomly chosen boolean *)

module G1 = {
  proc f() : bool = {
    var x : bool;
    x <$ {0,1};        (* sample x in {0,1} *)
    return x;
  }
}.

(* G2.f() yields the exclusive-or of two randomly chosen
   booleans *)

module G2 = {
  proc f() : bool = {
    var x, y : bool;
    x <$ {0,1}; y <$ {0,1};
    return x ^^ y;
  }
}.

(* PRHL judgement relating G1.f and G2.f. ={res} means res{1} =
   res{2}, i.e., the result of G1.f is equal to (has same distribution
   as) result of G2.f *)

lemma G1_G2_f : equiv[G1.f ~ G2.f : true ==> ={res}].
proof.
  proc.
  (* handle choice of x in G2.f *)
  seq 0 1 : true.
    rnd {2}. skip. smt.
  (* handle choice of x in G1.f / y in G2.f *)
  rnd (fun (z : bool) => z ^^ x{2}). skip. smt.
qed.

(* G1.f and G2.f are equally likely to return true: *)

lemma G1_G2_true &m :
  Pr[G1.f() @ &m : res] = Pr[G2.f() @ &m : res].
proof.
  byequiv.
  apply G1_G2_f. trivial. trivial.
```

```
qed.

(* G1.f and G2.f are equally likely to return false: *)

lemma G1_G2_false &m :
  Pr[G1.f() @ &m : !res] = Pr[G2.f() @ &m : !res].
proof.
  byequiv.
  apply G1_G2_f. trivial. trivial.
qed.
```

Listing 1.1: Sample EasyCrypt Script

As can be inferred from the example, comments begin and end with `(*` and `*)`, respectively; they may be nested. Each sentence of an EasyCrypt script is terminated with a dot (period, full stop). Much can be learned by experimenting with this script, and in particular by executing it step-by-step in Emacs.

1.4 More Information

More information about EasyCrypt—and about the EasyCrypt Team and its work—may be found at

https://www.easycrypt.info

The EasyCrypt Club mailing list features discussion about EasyCrypt usage:

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

Support requests should be sent to this list, as answers to questions will be of use to other members of the EasyCrypt community.

1.5 Bug Reporting

EasyCrypt bugs should be reported using the Tracker:

https://www.easycrypt.info/trac/report

You can log into the Tracker to create tickets or comment on existing ones using any GitHub account.

1.6 About this Documentation

The source for this document, along with the macros and language definitions used, are available from its GitHub repository.[2] Feel free to use the language definitions to typeset your EasyCrypt-related documents, and to contribute improvements to the macros if you have any.

This document is intended as a reference manual for the EasyCrypt tool, and not as a tutorial on how to build a cryptographic proof, or how to conduct interactive proofs. We provide some detailed examples in Chapter 7, but they may still seem obscure even with a good understanding of cryptographic theory. We recommend experimenting.

[2] https://github.com/EasyCrypt/easycrypt-doc

Chapter 2

Specifications

In this chapter, we present EasyCrypt's language for writing cryptographic specifications. We start by presenting its typed expression language, go on to consider its module language for expressing cryptographic games, and conclude by presenting its ambient logic—which includes judgments of the HL, pHL and pRHL logics.

EasyCrypt has a typed expression language based on the polymorphic typed lambda calculus. Expressions are guaranteed to terminate, although their values may be under-specified. Its type system has:

- several pre-defined base types;
- product (tuple) and record types;
- user-defined abbreviations for types and parameterized types; and
- user-defined concrete datatypes (like lists and trees).

In its expression language:

- one may use operators imported from the EasyCrypt library, e.g., for the pre-defined base types;
- user-defined operators may be defined, including by structural recursion on concrete datatypes.

For each type, there is a type of probability distributions over that type.

EasyCrypt's modules consist of typed global variables and procedures. The body of a procedure consists of local variable declarations followed by a sequence of statements:

- ordinary assignments;
- random assignments, assigning values chosen from distributions to variables;
- procedure calls, whose results are assigned to variables;
- conditional (if-then-else) statements;
- while loops; and
- return statements (which may only appear at the end of procedures).

A module's procedures may refer to the global variables of previously declared modules. Modules may be nested. Modules may be parameterized by abstract modules, which may be used to model adversaries; and modules types—or interfaces—may be formalized, describing modules with at least certain specified typed procedures.

EasyCrypt has four logics: a probabilistic, relational Hoare logic (pRHL), relating pairs of procedures; a probabilistic Hoare logic (pHL) allowing one to carry out proofs about the probability of a procedure's execution resulting in a postcondition holding; an ordinary (possibilistic) Hoare logic (HL); and an ambient higher-order logic for proving general mathematical facts, as well as for connecting judgments from the other logics

Proofs are carried out using tactics, which is the focus of Chapter 3. EasyCrypt also has ways (theories and sections) of structuring specifications and proofs, which will be described in Chapter 4. In Chapter 5, we'll survey the EasyCrypt Library, which consists of numerous theories, defining mathematical structures (like groups, rings and fields), data structures (like finite sets and maps), and cryptographic constructions (like random oracles and different forms of encryption).

2.1 Lexical Categories

EasyCrypt's language has a number of lexical categories:

- **Keywords**. EasyCrypt has the following *keywords*: **abbrev**, **abort**, **abstract**, **admit**, **admitted**, **algebra**, **alias**, **apply**, **as**, **assert**, **assumption**, **async**, **auto**, **axiom**, **axiomatized**, **beta**, **by**, **byequiv**, **byphoare**, **bypr**, **call**, **case**, **cfold**, **change**, **class**, **clear**, **clone**, **congr**, **conseq**, **const**, **cut**, debug, **declare**, **delta**, **do**, **done**, **dump**, **eager**, **elif**, **elim**, **else**, **end**, **equiv**, **eta**, **exact**, **exfalso**, **exists**, **expect**, **export**, **fel**, **field**, **fieldeq**, **first**, **fission**, **forall**, **fun**, **fusion**, **glob**, **goal**, **have**, **hint**, **hoare**, **idtac**, **if**, **import**, **in**, **include**, **inductive**, **inline**, **instance**, **iota**, **islossless**, **kill**, **last**, **left**, **lemma**, **let**, **local**, **logic**, **modpath**, **module**, **move**, **nosmt**, **notation**, **of**, **op**, **phoare**, **pose**, **Pr**, **pr_bounded**, pragma, **pred**, **print**, **proc**, **progress**, **proof**, **prover**, **qed**, **rcondf**, **rcondt**, **realize**, **reflexivity**, **remove**, **rename**, **replace**, **require**, **res**, **return**, **rewrite**, **right**, **ring**, **ringeq**, **rnd**, **rwnormal**, **search**, **section**, **Self**, **seq**, **sim**, **simplify**, **skip**, **smt**, **solve**, **sp**, **split**, **splitwhile**, **strict**, **subst**, **suff**, **swap**, **symmetry**, **then**, **theory**, time, **timeout**, **Top**, **transitivity**, **trivial**, **try**, **type**, undo, **unroll**, **var**, **while**, **why3**, **with**, **wlog**, **wp** and **zeta**.
- **Identifiers**. An *identifier* is a sequence of letters, digits, underscores (_) and apostrophes (') that begins with a letter or underscore, and isn't equal to an underscore or a keyword other than **abort**, **admitted**, **async**, **dump**, **expect**, **field**, **fieldeq**, **first**, **last**, **left**, **right**, **ring**, **ringeq**, **solve**, **strict** or **wlog**.
- **Operator names**. An *operator name* is an identifier, a binary operator name, a unary operator name, or a mixfix operator name.
- **Binary operator names**. A *binary operator name* is:
  - a nonempty sequence of equal signs (=), less than signs (<), greater than signs (>), forward slashes (/), backward slashes (\), plus signs (+), minus signs (-), times signs (*), vertical bars (|), colons (:), ampersands (&), up arrows (^) and percent signs (%); or
  - a backtick mark (`), followed by a nonempty sequence of one of these characters, followed by a backtick mark; or
  - a backward slash followed by a nonempty sequence of letters, digits, underscores and apostrophes.

  A binary operator name is an *infix operator name* iff it is surrounded by backticks, or begins with a backslash, or:

  - it is neither << nor >>; and
  - it doesn't contain a colon, unless it is a sequence of colons of length at least two; and
  - it doesn't contain =>, except if it is =>; and
  - it doesn't contain |, except if it is ||; and
  - it doesn't contain /, except if it is /, /\, or a sequence of slashes of length at least 3.

The precedence hierarchy for infix operators is (from lowest to highest):

  - => (right-associative);
  - <=> (non-associative);
  - || and \/ (right-associative);
  - && and /\ (right-associative);
  - = and <> (non-associative);
  - <, >, <= and >= (left-associative);
  - - and + (left-associative);
  - *, and any nonempty combination of / and % (other than //, which is illegal) (left-associative);
  - all other infix operators except sequences of colons (left-associative);
  - sequences of colons of length at least two (right-associative).

- **Unary operator names**. A *unary operator name* is a negation sign (!), a nonempty sequence of plus signs (+), a nonempty sequence of minus signs (-), or a backward slash followed by a nonempty sequence of letters, digits, underscores and apostrophes. A *prefix operator name* is any unary operator name not consisting of either two more plus signs or two or more minus signs.

- **Mixfix operator names**. A *mixfix operator name* is of the following sequences of characters: `|_|, [], _.[_] or _.[_<-_]. (We'll see below how they may be used in mixfix form.)

- **Record field projections**. A *record field projection* is an identifier.

- **Constructor names**. A *constructor name* is an identifier or a symbolic operator name.

- **Type variables**. A *type variable* consists of an apostrophe followed by a sequence of letters, digits, underscores and apostrophes that begins with a lowercase letter or underscore, and isn't equal to an underscore.

- **Type or type operator names**. A *type or type operator name* is an identifier.

- **Variable names**. A *variable* name is an identifier that doesn't begin with an uppercase letter.

- **Procedure names**. A *procedure* name is an identifier that doesn't begin with an uppercase letter.

- **Module names**. A *module name* is an identifier that begins with an uppercase letter.

- **Module type names**. A *module type name* is an identifier that begins with an uppercase letter.

- **Lemma and axiom names**. A *lemma or axiom name* is an identifier.

- **Theory names**. A *theory name* is an identifier that begins with an uppercase letter.

- **Section names**. A *section name* is an identifier that begins with an uppercase letter.

- **Memory identifiers**. A *memory identifier* consists an ampersand followed by either a nonempty sequence of digits or an identifier whose initial character isn't an upper case letter.

2.2 Script Structure, Name spaces, Printing and Searching

An EasyCrypt script consists of a sequence of *steps*, terminated by dots (.). Steps may:

- declare types;
- declare operators and predicates;
- declare modules or module types;
- state axioms or lemmas;
- apply tactics;
- require (make available) theories;
- enter or exit theories or sections;
- print types, operators, predicates, modules, module types, axioms and lemmas;
- search for lemmas involving operators.

Operators, predicates, record field projections and type constructors share the same name space. Lemmas and axioms share the same name space.

To print an entity, one may say:

```
print type t.
print op f.
print op (+).
print op [-].
print op "_.[_]".
print pred p.
print module Foo.
print module type FOO.
print axiom foo.
print lemma goo.
```

The entity kind may be omitted, in which case all entities with the given name are printed. `print op` and `print pred` may be used interchangeably, and may be applied to record field projections and datatype constructors, as well as to operators and predicates. `print axiom` and `print lemma` are also interchangeable. Infix operators must be parenthesized; unary operators must be enclosed in square brackets; and mixfix operators must be enclosed in double quotation marks.

To search for axioms and lemmas involving all of a list of operators, one can say

```
search f.
search (+).
search [-].
search "_.[_]".
search (+) (-).  (* axioms/lemmas involving both operators *)
```

Declared/stated entities may refer to previously declared/stated entities, but not to themselves or later ones (with the exception of recursively declared operators on datatypes, and to references to a module's own global variables).

2.3 Expressions Language

2.3.1 Type Expressions

EasyCrypt's *type expressions* are built from *type variables*, *type constructors* (or *named types*) function types and tuple (product) types. Type constructors include built-in types and user-defined types, such as *record* types and *datatypes* (or *variant types*). The syntax of type expressions

τ, σ	::=	`tyvar`	**type variable**
		`_`	**anonymous type variable**
		`(`τ`)`	**parenthesized type**
		τ `->` σ	**function type**
		`(`τ_1 `*` $\cdots$ `*` τ_n`)`	**tuple type**
		`tyname`	**named type**
		τ `tyname`	**applied type constructor**
		`(`τ_1`,` `...`, τ_n`)` `tyname`	***ibid.***

Figure 2.1: EasyCrypt's type expressions

	Operator	Associativity
	type constructor application	—
`*`	tuple constructor	—
`->`	function type	right

constructions with higher precedences come first

Figure 2.2: Type operators precedence and associativity

is given in Figure 2.1, whereas the precedence and associativity of type operators are given in Figure 2.2.

It is worth noting that EasyCrypt's types must be inhabited — i.e. nonempty.

Built-in types EasyCrypt comes with built-in types for booleans (`bool`), integers (`int`) and reals (`real`), along with the singleton type `unit` that is inhabited by the single element `tt` (or `()`).

In addition, to every type t is associated the type t `distr` of *(real) discrete sub-distribution*. A discrete sub-distribution over a type t is fully defined by its mass function, i.e. by a non-negative function from t to $\mathbb{R}$ s.t. $\sum_x f(x) \leq 1$ — implying that f has a discrete support. When the sum is equal to 1, we say that we have a *distribution*. Note that `distr` is not a proper type on its own, but a *type constructor*, i.e. a function from types to types. A proper type is obtained by *applying* `distr` to an actual type, as in `int distr` or `bool distr`. See the paragraph on type constructors for more information.

Function types The type expression τ `->` σ denotes the type of *total functions* mapping elements of type τ to elements of type σ. Note that `->` associates to the right, so that `int -> bool -> real` and `int -> (bool -> real)` denotes the same type.

Tuple (product) types The type expression τ_1 `*` $\cdots$ `*` τ_n denotes the type of n-tuples whose elements are resp. of type τ_i. This includes the type of pairs as well as the type of tuples of 3 elements or more. Note that τ_1 `* (`τ_2 `*` τ_3`)`, `(`τ_1 `*` τ_2`)` `*` τ_3 and τ_1 `*` τ_2 `*` τ_3 are all distinct types. The first two are pair types, whereas the last one is the type of 3-tuples.

Type variables Type variables represent unknown types or type parameters. For example, the type `'a * 'a` is the type the of pair whose elements are of unknown type `'a`. Type variables may be used in type declarations (Section 2.3.2) to define type constructors or in operators/predicates declarations (Section 2.3.3) to define polymorphic operators/predicates. The special type variable `_` (underscore) represents a type variable whose name is not specified.

Type constructors Type constructors are not type expressions per se, but functions from types to types. As seen in the built-in section, `distr` is such a type constructor: when applied to the type τ, it gives the type τ `distr` of sub-distributions over τ. Note that the application is in *postfix* form. One other common type constructors is the one `list` of polymorphic list, the type expression τ `list` denoting the type of lists whose elements are of type τ.

Type constructors may depend on several type arguments, i.e. may be of arity strictly greater than 1. In that case, the type application is curried. For example, the type of finite map (τ, σ) `map` (whose keys are of type τ and values of type σ) is constructed from the type constructor `map` of arity 2.

By abuse of notations, named types (as `bool` or `int`) can be seen as type constructors with no arguments.

Datatypes and record types There are no expressions for describing datatypes and record types. Indeed, those are always named and must be defined and named before use. See Section 2.3.2 for how to define variant and record types.

2.3.2 Type Declarations

Record types may be declared like this:

```
type t = { x : int; y : bool; }.
type u = { y : real; yy : int; yyy : real; }.
```

Here `t` is the type of records with field projections `x` of type `int`, and `y` of type `bool`. The order of projections is irrelevant. Different record types can't use overlapping projections, and record projections must be disjoint from operators (see below). Records may have any non-zero number of fields; values of type `u` are record with three fields. We may also define record type operators, as in:

```
type 'a t = { x : 'a; f : 'a -> 'a; }.
type ('a, 'b) u = { f : 'a -> 'b; x : 'a; }.
```

Then, a value v of type `int t` would have fields `x` and `f` of types `int` and `int -> int`, respectively; and a value v of type `(int, bool) u` would have fields `x` and `f` with types `int` and `int -> bool`, respectively.

Datatypes and datatype operators may be declared like this:

```
type enum = [ First | Second | Third ].
type either_int_bool = [ First of int | Second of bool ].
type ('a, 'b) either = [ First of 'a | Second of 'b ].
type intlist = [
  | Nil
  | Cons of (int * intlist) ].
type 'a list = [
  | Nil
  | Cons of 'a & 'a list ].
```

Here, `First`, `Second`, `Third`, `Nil` and `Cons` are constructors, and must be distinct from all operators, record projections and other constructors. `enum` is an enumerated type with the three elements `First`, `Second` and `Third`. The elements of `either_int_bool` consist of `First` applied to an integer, or `Second` applied to a boolean, and the datatype operator `either` is simply its generalization to arbitrary types `'a` and `'b`. `intlist` is an inductive datatype: its elements are `Nil` and the results of applying `Cons` to a pairs of the form (x, ys), where x is an integer and ys is a previously constructed `intlist`. Note that a vertical bar (`|`) is permitted before the first constructor of a datatype. Finally, `list` is the generalization of `intlist` to lists over an arbitrary type `'a`, but with a twist. The use of `&` means that `Cons` is "curried": instead of applying `Cons` to a pair (x, ys), one gives it x `: 'a` and ys `: 'a list` one at a time, as in `Cons` x ys. Unsurprisingly, more than one occurrence of `&` is allowed in a constructor's definition. E.g., here is the datatype for binary trees whose leaves and internal nodes are labeled by integers:

```
type tree = [
  | Leaf of int
```

```
  | Cons of tree & int & tree
].
```

`Cons` $tr_1\, x\, tr_2$ will be the tree constructed from an integer x and trees tr_1 and tr_2. EasyCrypt must be able to convince itself that a datatype is nonempty, most commonly because it has at least one constructor taking no arguments, or only arguments not involving the datatype.

Types and type operators that are simply abbreviations for pre-existing types may be declared, as in:

```
type t = int * bool.
type ('a, b) arr = 'a -> 'b.
```

Then, e.g., `(int, bool) arr` is the same type as `int -> bool`.

Finally, abstract types and type operators may be declared, as in:

```
type t.
type ('a, b) u.
type t, ('a, b) u.
```

We'll see later how such types and type operators may be used.

2.3.3 Expressions and Operator Declarations

We'll now survey EasyCrypt's typed expressions. Anonymous functions are written

$$\texttt{fun}\ (x : t_1)\ \texttt{=>}\ e,$$

where x is an identifier, t_1 is a type, and e is an expression—probably involving x. If e has type t_2 under the assumption that x has type t_1, then the anonymous function will have type t_1 `->` t_2. Function application is written using juxtapositioning, so that if e_1 has type t_1 `->` t_2, and e_2 has type t_1, then $e_1\, e_2$ has type t_2. Function application associates to the left, and anonymous functions extend as far to the right as possible. EasyCrypt infers the types of the bound variables of anonymous function when it can. Nested anonymous functions may be abbreviated by collecting all their bound variables together. E.g., consider the expression

```
(fun (x : int) => fun (y : int) => fun (z : bool) => y) 0 1 false
```

which evaluates to `1`. It may be abbreviated to

```
(fun (x y : int, z : bool) => y) 0 1 false
```

or

```
(fun (x : int) (y : int) (z : bool) => y) 0 1 false
```

or (letting EasyCrypt carry out type inference)

```
(fun x y z => y) 0 1 false
```

In the type inference, only the type of `y` is determined, but that's acceptable.

EasyCrypt has let expressions

$$\texttt{let}\ x : \texttt{t} = e\ \texttt{in}\ e$$

which are equivalent to

$$(\texttt{fun}\ x : \texttt{t} \texttt{=>}\ e)e$$

As with anonymous expressions, the types of their bound variables may often be omitted, letting EasyCrypt infer them.

An operator may be declared by specifying its type and giving the expression to be evaluated. E.g.,

```
op x : int = 3.
op f : int -> bool -> int = fun (x : int) (y : bool) => x.
op g : bool -> int = f 1.
op y : int = g true.
op z = f 1 true.
```

Here `f` is a curried function—it takes its arguments one at a time. Hence `y` and `z` have the same value: `1`. As illustrated by the declaration of `z`, one may omit the operator's type when it can be inferred from its expression. The declaration of `f` may be abbreviated to

```
op f (x : int) (y : bool) = x.
```

or

```
op f (x : int, y : bool) = x.
```

Polymorphic operators may be declared, as in

```
op g ['a, 'b] : 'a -> 'b -> 'a = fun (x : 'a, y : 'b) => x.
```

or

```
op g ['a, 'b] (x : 'a, y : 'b) = x.
```

or

```
op g (x : 'a, y : 'b) = x.
```

Here `g` has all the types formed by substituting types for the types variable `'a` and `'b` in `'a -> 'b -> 'a`. This allows us to use `g` at different types

```
op a = g true 0.
op b = g 0 false.
```

making `a` and `b` evaluate to `true` and `0`, respectively.

Abstract operators may be declared, i.e., ones whose values are unspecified. E.g., we can declare

```
op x : int.
op f : int -> int.
op g ['a, 'b] : 'a -> 'b -> 'a.
```

Equivalently, `f` and `g` may be declared like this:

```
op f (x : int) : int.
op g ['a, 'b] (x : 'a, y : 'b) : 'a.
```

One may declare multiple abstract operators of the same type:

```
op f, g : int -> int.
op g, h ['a, 'b] : 'a -> 'b -> 'a.
```

We'll see later how abstract operators may be used.

Binary operators may be declared and used with infix notation (as long as they are infix operators). One parenthesizes a binary operator when declaring it and using it in non-infix form (i.e., as a value). If io is an infix operator and e_1, e_2 are expressions, then $e_1\ io\ e_2$ is translated to $(io)\ e_1\ e_2$, whenever the latter expression is well-typed. E.g., if we declare

```
op (--) ['a, 'b] (x : 'a) (y : 'b) = x.
op x : int = (--) 0 true.
op x' : int = 0 -- true.
op y : bool = (--) true 0.
op y' : bool = true -- 0.
```

then `x` and `x'` evaluate to `0`, and `y` and `y'` evaluate to `true`.

Unary operators may be declared and used with prefix notation (as long as they are prefix operators). One (square) brackets a unary operator when declaring it and using it in non-prefix form (i.e., as a value). If *po* is a prefix operator and *e* is an expression, then *po e* is translated to `[`*po*`]` *e*, whenever the latter expression is well-typed. E.g., if we declare

```
op x : int.
op f : int -> int.
op [!] : int -> int.
op y : int = ! f x.
op y' : int = [!](f x).
```

then `y` and `y'` both evaluate to the result of applying the abstract operator `!` of type `int -> int` to the result of applying the abstract operator `f` of type `int -> int` to the abstract value `x` of type `int`. Function application has higher precedence than prefix operators, which have higher precedence than infix operators, prefix operators group to the right, and infix operators have the associativities and relative precedences that were detailed in Section 2.1.

The four mixfix operators may be declared and used as follows. They are (double) quoted when being declared or used in non-mixfix form (i.e., as values).

- (`[]`) `[]` is translated to `"[]"`. E.g., if we declare

  ```
  op "[]" : int = 3.
  op x : int = [].
  ```

  then `x` will evaluate to `3`.

- (`` `|_| ``) If *e* is an expression, then `` `| ``*e*`|` is translated to `` "`|_|" ``*e*, as long as the latter expression is well-typed. E.g., if we declare

  ```
  op "`|_|" : int -> bool.
  op x : bool = "`|_|" 3.
  op y : bool = `|3|.
  ```

  then `y` will evaluate to the same value as `x`.

- (`_.[_]`) If e_1, e_2 are expressions, then e_1`.[`e_2`]` is translated to `"_.[_]"` $e_1\, e_2$, whenever the latter expression is well-typed. E.g., if we declare

  ```
  op "_.[_]" : int -> int -> bool.
  op x : bool = "_.[_]" 3 4.
  op y : bool = 3.[4].
  ```

  then `y` will evaluate to the same value as `x`.

- (`_.[_<-_]`) If e_1, e_2, e_3 are expressions, e_1`.[`e_2 `<-` e_3`]` is translated to `"_.[_<-_]"` $e_1\, e_2\, e_3$, whenever the latter expression is well-typed. E.g., if we declare

  ```
  op "_.[_<-_]" : int -> int -> int -> bool.
  op x : bool = "_.[_<-_]" 3 4 5.
  op y : bool = 3.[4 <- 5].
  ```

  then `y` will evaluate to the same value as `x`.

In addition, if $e_1, \ldots, e_n$ are expressions then

`[`e_1`;` $\ldots$ `;` e_n`]` is translated to e_1 `::` $\ldots$ `::` e_n `::` `[]`

whenever the latter expression is well-typed. The initial argument of `"_.[_]"` and `"_.[_<-_]"` have higher precedence than even function application. E.g., one can't omit the parentheses in

```
op f : int -> int.
op y : bool = (f 3).[4].
op z : bool = (f 3).[4 <- 5].
```

Some operators are built-in to EasyCrypt, automatically understood by its ambient logic:

```
op (=) ['a]: 'a -> 'a -> bool.

op [!] : bool -> bool.
op (||) : bool -> bool -> bool.
op (\/) : bool -> bool -> bool.
op (&&) : bool -> bool -> bool.
op (/\) : bool -> bool -> bool.
op (=>) : bool -> bool -> bool.
op (<=>) : bool -> bool -> bool.

op mu : 'a distr -> ('a -> bool) -> real.
```

The operator `=` is equality. On the booleans, we have negation `!`, two forms of disjunction (`\/` and `||`) and conjunction (`/\` and `&&`), implication (`=>`) and if-and-only-if (`<=>`). The two disjunctions (respectively, conjunctions) are semantically equivalent, but are treated differently by EasyCrypt proof engine. The associativities and precedences of the infix operators were given in Section 2.1, and (as a prefix operator) `!` has higher precedence than all of them. The expression e_1 `<>` e_2 is treated as `!(`e_1 `=` e_2`)`. `<>` is not an operator, but it has the precedence and non-associative status of Section 2.1. The intended meaning of `mu` $d\,p$ is the probability that randomly choosing a value of the given type from the sub-distribution d will satisfy the function p (in the sense of causing it to return `true`).

If e is an expression of type `int`, then e%r is the corresponding real. _%r has higher precedence than even function application.

If e_1 is an expression of type `bool` and e_2, e_3 are expressions of some type t, then the *conditional expression*

$$e_1 \ ?\ e_2 \ :\ e_3$$

evaluates to e_2, if e_1 evaluates to `true`, and evaluates to e_3, if e_1 evaluates to `false`. Conditionals may also be written using if-then-else notation:

if e_1 **then** e_2 **else** e_3

E.g., if we write

```
op x : int = (3 < 4) ? 4 + 7 : (9 - 1).
```

then `x` evaluates to `11`. The conditional expression's precedence at its first argument is lower than function application, but higher than the prefix operators; its second argument needn't be parenthesized; and the precedence at its third argument is lower than the prefix operators, but higher than the infix operators.

For the built-in types `bool`, `int` and `real`, and the type operator `distr`, the EasyCrypt Library (see Chapter 5) provides corresponding theories, `Bool`, `Int`, `Real` and `Distr`. These theories provide various operations, axioms, etc. To make use of a theory, one must "require" it. E.g.,

```
require Bool Int Real Distr.
```

will make the theories just mentioned available. This would allow us to write, e.g.,

```
op x = Int.(+) 3 4.
```

making `x` evaluate to `7`. But to be able to use `+` and the other operators provided by `Int` in infix form and without qualification (specifying which theory to find them in), we need to import `Int`. If we do

```
import Bool Int Real.
op x : int = 3 + 4 - 7 * 2.
op y : real = 5%r * 3%r / 2%r.
op z : bool = x%r >= y.
```

we'll end up with `z` evaluating to `false`. One may combine requiring and importing in one step:

```
require import Bool Int Real Distr.
```

We'll cover theories and their usage in detail in Chapter 4.

Requiring the theory `Bool` makes available the value `{0,1}` of type `bool distr`, which is the uniform distribution on the booleans. (No whitespace is allowed in the name for this distribution, and the `0` must come before the `1`.) Requiring the theory `Distr` make available syntax for the uniform distribution of integers from a finite range. If e_1 and e_2 are expressions of type `int` denoting n_1 and n_2, respectively, then `[`e_1`..`e_2`]` is the value of type `int distr` that is the uniform distribution on the set of all integers that are greater-than-or-equal to n_1 and less-than-or-equal-to n_2—unless $n_1 > n_2$, in which case it is the sub-distribution assigning probability 0 to all integers.

Values of product (tuple) and record types are constructed and destructed as follows:

```
op x : int * int * bool = (3, 4, true).
op b : bool = x .` 3.
type t = { u : int; v : bool; }.
op y : t = {| v = false; u = 10; |}.
op a : bool = y .` v.
```

Then, `b` evaluates to `true`, and `a` evaluates to `false`. Note the field order in the declaration of `y` was allowed to be a permutation of that of the record type `t`.

When we declare a datatype, its constructors are available to us as values. E.g, if we declare

```
type ('a, 'b) either = [Fst of 'a | Snd of 'b].
op x : (int, bool) either = Fst 10.
op y : int -> (int, bool) either = Fst.
op z : (int, bool) either = y 10.
```

then `z` evaluates to the same result as `x`.

We can declare operators using pattern matching on the constructors of datatypes. E.g., continuing the previous example, we can declare and use an operator `fst` by:

```
op fst ['a, 'b] (def : 'a) (ei : ('a, 'b) either) : 'a =
  with ei = Fst a => a
  with ei = Snd b => def.
op l1 : (int, bool) either = Fst 10.
op l2 : (int, bool) either = Snd true.
op m1 : int = fst (-1) l1.
op m2 : int = fst (-1) l2.
```

Here, `m1` will evaluate to `10`, whereas `m2` will evaluate to `-1`. Such operator declarations may be recursive, as long as EasyCrypt can determine that the recursion is well-founded. E.g., here is one way of declaring an operator `length` that computes the length of a list:

```
type 'a list = [Nil | Cons of 'a & 'a list].
op len ['a] (acc : int, xs : 'a list) : int =
  with xs = Nil => acc
  with xs = Cons y ys => len (acc + 1) ys.
op length ['a] (xs : 'a list) = len 0 xs.
op xs = Cons 0 (Cons 1 (Cons 2 Nil)).
op n : int = length xs.
```

Then `n` will evaluate to `3`.

2.4 Module System

2.4.1 Modules

EasyCrypt's modules consist of typed global variables and procedures, which have different name spaces. Listing 2.1 contains the definition of a simple module, `M`, which exemplifies much of the module language.

```
require import Int Bool DInterval.

module M = {
  var x : int

  proc init(bnd : int) : unit = {
    x <$ [-bnd .. bnd];
  }

  proc incr(n : int) : unit = {
    x <- x + n;
  }

  proc get() : int = {
    return x;
  }

  proc main() : bool = {
    var n : int;
    init(100);
    incr(10); incr(-50);
    n <@ get();
    return n < 0;
  }
}.
```

Listing 2.1: Simple Module

`M` has one *global variable*—`x`—which is used by the *procedures* of `M`—`init`, `incr`, `get` and `main`. Global variables must be declared before the procedures that use them.

The procedure `init` ("initialize") has a *parameter* (or *argument*) `bnd` ("bound") of type `int`. `init` uses a *random assignment* to assign to `x` an integer chosen uniformly from the integers whose absolute values are at most `bnd`. The return type of `init` is `unit`, whose only element is `tt`; this is implicitly returned by `init` upon exit.

The procedure `incr` ("increment"), increments the value of `x` by its parameter `n`. The procedure `get` takes no parameters, but simply returns the value of `x`, using a *return statement*—which is only allowed as the final statement of a procedure.

And the `main` procedures takes no parameters, and returns a boolean that's computed as follows:

- It declares a local variable, `n`, of type `int`—local in the sense that other procedures can't access or affect it.
- It uses a *procedure call* to call the procedure `init` with a bound of `100`, causing `x` to be initialized to an integer between `-100` and `100`.
- It calls `incr` twice, with `10` and then `-50`.
- It uses a *procedure call assignment* to call the procedure `get` with no arguments, and assign `get`'s return value to `n`.
- It evaluates the boolean expression `n < 0`, and returns the value of this expression as its boolean result.

EasyCrypt tries to infer the return types of procedures and the types of parameters and local variables. E.g., our example module could be written

```
module M = {
  var x : int

  proc init(bnd) = {
    x <$ [-bnd .. bnd];
  }

  proc incr(n) = {
    x <- x + n;
  }

  proc get() = {
    return x;
  }

  proc main() = {
    var n;
    init(100);
    incr(10); incr(-50);
    n <@ get();
    return n < 0;
  }
}.
```

Listing 2.2: Simple Module with Type Inference

As we've seen, each declaration or statement of a procedure is terminated with a semicolon. One may combine multiple local variable declarations, as in:

```
var x, y, z : int;
var u, v;
var x, y, z : int <- 10;
var x, y, z <- 10;
```

Procedure parameters are variables; they may be modified during the execution of their procedures. A procedure's parameters and local variables must be distinct variable names. The three kinds of assignment statements differ according to their allowed right-hand sides (rhs):

- The rhs of a random assignment must be a single (sub-)distribution. When choosing from a proper sub-distribution, the random assignment may fail, causing the procedure call that invoked it to fail to terminate.

- The rhs of an ordinary assignment may be an arbitrary expression (which doesn't include use of procedures).

- the rhs of a procedure call assignment must be a single procedure call.

If the rhs of an assignment produces a tuple value, its left-hand side may use pattern matching, as in

```
(x, y, z) <- ...;
```

in the case where ... produces a triple.

The two remaining kinds of statements are illustrated in Listing 2.3: *conditionals* and *while loops*.

```
require import Bool Int DInterval.

module N = {
```

```
    proc loop() : int = {
      var y, z : int;
      y <- 0;
      while (y < 10) {
        z <$ [1 .. 10];
        if (z <= 5) {
          y <- y - z;
        }
        else {
          y <- y + (z - 5);
        }
      }
      return y;
    }
  }.
```

Listing 2.3: Conditionals and While Loops

`N` has a single procedure, `loop`, which begins by initializing a local variable `y` to `0`. It then enters a while loop, which continues executing until (which may never happen) `y` becomes `10` or more. At each iteration of the while loop, an integer between `1` and `10` is randomly chosen and assigned to the local variable `z`. The conditional is used to behave differently depending upon whether the value of `z` is less-than-or-equal-to `5` or not.

- When the answer is "yes", `y` is decremented by `z`.
- When the answer is "no", `y` is incremented by `z - 5`.

Once (if) the while loop is exited—which means `y` is now `10` or more—the procedure returns `y`'s value as its return value.

When the body of a while loop, or the then or else part of a conditional, has a single statement, the curly braces may be omitted. E.g., the conditional of the preceding example could be written:

```
if (z <= 5) y <- y - z;
else y <- y + (z - 5);
```

And when the else part of a conditional is empty (consists of `{}`), it may be omitted, as in:

```
if (z <= 5) y <- y - z;
```

As illustrated in Listing 2.4, modules may access the global variables, and call the procedures, of previously declared modules.

```
require import Bool Int.

module M = {
  var x : int

  proc f() : unit = {
    x <- x + 1;
  }
}.

module N = {
  var x : int

  proc g(n m : int, b : bool) : bool = {
    if (b) M.f();
    M.x <- M.x + x + n - m;
    return 0 < M.x;
  }
```

```
  proc h = M.f
}.
```

Listing 2.4: One Module Using Another Module

Procedure `g` of `N` both accesses the global variable `x` of module `M` (`M.x`), and calls `M`'s procedure, `f` (`M.f`). The parameter list of `g` could equivalently be written:

```
n : int, m : int, b : bool
```

A module may refer to its own global variables using its own module name, allowing us to write

```
proc f() : unit = {
  M.x <- M.x + 1;
}
```

for the definition of procedure `M.f`. The procedure `h` of `N` is an alias for procedure `M.f`: calling it is equivalent to directly calling `M.f`. One declare a module name to be an alias for a module, as in

```
module L = N.
```

A procedure call is carried out in the context of a *memory* recording the values of all global variables of all declared modules. So all global variables are—by definition—initialized. On the other hand, the local variables of a procedure start out as arbitrary values of their types. This is modeled in EasyCrypt's program logics by our not knowing anything about them. For example, the probability of `X.f()`

```
module X = {
  proc f() : bool = {
    var b : bool;
    return b;
  }
}.
```

returning `true` is undefined—we can't prove anything about it. On the other hand, just because a local variable isn't initialized before use doesn't mean the result of its use will be indeterminate, as illustrated by the procedure `Y.f`, which always returns `0`:

```
module Y = {
  proc f() : int = {
    var x : int;
    return x - x;
  }
}.
```

2.4.2 Module Types

EasyCrypt's *module types* specify the types of a set of procedures. E.g., consider the module type `OR` :

```
module type OR = {
  proc init(secret : int, tries : int) : unit
  proc guess(guess : int) : unit
  proc guessed() : bool
}.
```

`OR` describes minimum expectations for a "guessing oracle"—that it provide at least procedures with the specified types. The order of the procedures in a module type is irrelevant. In a procedure's type, one may combine multiple parameters of the same type, as in:

```
proc init(secret tries : int) : unit
```

The names of procedure parameters used in module types are purely for documentation purposes; one may elide them instead using underscores, writing, e.g.,

```
proc init(_ : int, _ : int) : unit
```

Note that module types say nothing about the global variables a module should have. Modules types have a different name space than modules.

Listing 2.5 contains an example guessing oracle implementation.

```
module Or = {
  var sec : int
  var tris : int
  var guessed : bool

  proc init(secret, tries : int) : unit = {
    sec <- secret;
    tris <- tries;
    guessed <- false;
  }

  proc guess(guess : int) : unit = {
    if (0 < tris) {
      guessed <- guessed \/ (guess = sec);
      tris <- tris - 1;
    }
  }

  proc guessed() : bool = {
    return guessed;
  }
}.
```

Listing 2.5: Guessing Oracle Module

Its `init` procedure stores the supplied secret in the global variable `sec`, initializes the allowed number of guesses in the global variable `tris`, and initializes the `guessed` global variable to record that the secret hasn't yet been guessed. If more allowed tries remain, the `guess` procedure updates `guessed` to take into account the supplied guess, and decrements the allowed number of tries; otherwise, it does nothing. And its `guessed` procedure returns the value of `guessed`, indicating whether the secret has been successfully guessed, so far. `Or` *satisfies* the specification of the module type `OR`, and we can ask EasyCrypt to check this by supplying that module type when declaring `Or`, as in Listing 2.6.

```
module Or : OR = {
  var sec : int
  var tris : int
  var guessed : bool

  proc init(secret tries : int) : unit = {
    sec <- secret;
    tris <- tries;
    guessed <- false;
  }

  proc guess(guess : int) : unit = {
    if (0 < tris) {
      guessed <- guessed \/ (guess = sec);
      tris <- tris - 1;
    }
  }
```

```
  proc guessed() : bool = {
    return guessed;
  }
}.
```

Listing 2.6: Guessing Oracle Module with Module Type Check

Supplying a module type *doesn't* change the result of a module declaration. E.g., if we had omitted `guessed` from the module type `OR`, the module `Or` would still have had the procedure `guessed`. Furthermore, when declaring a module, we can ask EasyCrypt to check whether it satisfies multiple module types, as in:

```
module type A = { proc f() : unit }.
module type B = { proc g() : unit }.
module X : A, B = {
  var x, y : int
  proc f() : unit = { x <- x + 1; }
  proc g() : unit = { y <- y + 1; }
}.
```

When declaring a module alias, one may ask EasyCrypt to check that the module matches a module type, as in:

```
module X' : A, B = X.
```

Suppose we want to declare a cryptographic game using our guessing oracle, parameterized by an adversary with access to the `guess` procedure of the oracle, and which provides two procedures—one for choosing the range in which the guessing game will operate, and one for doing the guessing. We'd like to write something like:

```
module type GAME = {
  proc main() : bool
}.

module Game(Adv : ADV) : GAME = {
  module A = Adv(Or)
  proc main() : bool = { ... }
}.
```

Thus, the module type `ADV` for adversaries must be parameterized by an implementation of `OR`. Given the adversary procedures we have in mind, the syntax for this is

```
module type ADV(O : OR) = {
  proc chooseRange() : int * int
  proc doGuessing() : unit
}.
```

But this declaration would give the adversary access to all of `O`'s procedures, which isn't what we want. Instead, we can write

```
module type ADV(O : OR) = {
  proc chooseRange() : int * int {}
  proc doGuessing() : unit {O.guess}
}.
```

meaning that `chooseRange` has no access to the oracle, and `doGuessing` may only call its `guess` procedure. Using this notation, our original attempt would have to be written

```
module type ADV(O : OR) = {
  proc chooseRange() : int * int {O.init O.guess O.guessed}
  proc doGuessing() : unit {O.init O.guess O.guessed}
}
```

Finally, we can specify that `chooseRange` must initialize all of the adversary's global variables (if any) by using a star annotation:

```
module type ADV(O : OR) = {
  proc * chooseRange() : int * int {}
  proc doGuessing() : unit {O.guess}
}.
```

The enforcement of such checks is not carried out by EasyCrypt's type checker, but by its logic (see the **apply** (p. 60) and **rewrite** (p. 62) tactics).

The full Guessing Game example is contained in Listing 2.7.

```
require import Bool Int IntDiv DInterval.

module type OR = {
  proc init(secret tries : int) : unit
  proc guess(guess : int) : unit
  proc guessed() : bool
}.

module Or : OR = {
  var sec : int
  var tris : int
  var guessed : bool

  proc init(secret tries : int) : unit = {
    sec <- secret;
    tris <- tries;
    guessed <- false;
  }

  proc guess(guess : int) : unit = {
    if (0 < tris) {
      guessed <- guessed \/ (guess = sec);
      tris <- tris - 1;
    }
  }

  proc guessed() : bool = {
    return guessed;
  }
}.

module type ADV(O : OR) = {
  proc * chooseRange() : int * int {}
  proc doGuessing() : unit {O.guess}
}.

module (SimpAdv : ADV) (O : OR) = {
  var range : int * int
  var tries : int

  proc chooseRange() : int * int = {
    range <- (1, 100);
    tries <- 10;
    return range;
  }

  proc doGuessing() : unit = {
    var x : int;
    while (0 < tries) {
```

```
        x <$ [range.`1 .. range.`2];
        O.guess(x);
        tries <- tries - 1;
      }
    }
  }.

module type GAME = {
  proc main() : bool
}.

module Game(Adv : ADV) : GAME = {
  module A = Adv(Or)

  proc main() : bool = {
    var low, high, tries, secret : int;
    var advWon : bool;
    (low, high) <@ A.chooseRange();
    if (high - low < 10)
      advWon <- false;
    else {
      tries <- (high - low + 1) %/ 10;  (* /% is integer division *)
      secret <$ [low .. high];
      Or.init(secret, tries);
      A.doGuessing();
      advWon <@ Or.guessed();
    }
    return advWon;
  }
}.

module SimpGame : GAME = Game(SimpAdv).
```

Listing 2.7: Full Guessing Game Example

`SimpAdv` is a simple implementation of an adversary. The inclusion of the constraint `SimpAdv : ADV` in `SimpAdv`'s declaration

```
module (SimpAdv : ADV) (O : OR) = ...
```

makes EasyCrypt check that `SimpAdv` implements the module type `ADV`: its implementation of `chooseRange` doesn't use `O` at all; its implementation of `doGuessing` doesn't use any of `O`'s procedures other than `guess`; and that `chooseRange` initializes `SimpAdv`'s global variables. Its `chooseRange` procedure chooses the range of `1` to `100`, and initializes global variables recording this range and the number of guesses it will make (see the code of `Game` to see why `10` is a sensible choice). The `doGuessing` procedure makes `10` random guesses. It would be legal for the parameter `O` in `SimpAdv`'s definition to be constrained to match a module type `T` providing a proper subset of `OR`'s procedures, but that would further limit what procedures of `O` `SimpAdv`'s procedures could call. On the other hand, it would be illegal for `T` to provide procedures not in `OR`.

Despite `SimpAdv` being a parameterized module, to refer to one of its global variables from another module one ignores the parameter, saying, e.g.,

```
module X = {
  proc f() : int = {
    return SimpAdv.tries;
  }
}.
```

On the other hand, to call one of `SimpAdv`'s procedures, one needs to specify which oracle parameter it will use, as in:

```
module X = {
  proc f() : unit = {
    SimpAdv(Or).doGuessing();
  }
}.
```

The module `Game` gives its adversary parameter, `Adv`, the concrete guessing oracle `Or`, calling the resulting module `A`. Its main function then uses `Or` and `A` to run the game.

- It calls `A`'s `chooseRange` procedure to get the adversary's choice of guessing range. If the range doesn't have at least ten elements, it returns `false` without doing anything else—the adversary has supplied a range that's too small.
- Otherwise, it uses `Or.init` to initialize the guessing oracle with a secret that's randomly chosen from the range, plus a number of allowed guesses that's one tenth of the range's size.
- It then calls `A.doGuessing`, allowing the adversary to attempt the guess the secret.
- Finally, it calls `Or.guessed` to learn whether the adversary has guessed the secret, returning this boolean value as its result.

Finally, the declaration

```
module SimpGame = Game(SimpAdv).
```

declares `SimpGame` to be the specialization of `Game` to our simple adversary, `SimpAdv`. When processing this declaration, EasyCrypt's type checker verifies that `SimpAdv` satisfies the specification `ADV`. The reader might be wondering what—if anything—prevents us writing a version of `SimpAdv` that directly accesses/calls the global variables and procedures of `Or` (or of `Game`, were `SimpAdv` declared after it), violating our understanding of the adversary's power. The answer is that EasyCrypt's type checker isn't in a position to do this. Instead, we'll see in the next section how such constraints are modeled using EasyCrypt's logic.

2.4.3 Global Variables

The set of all *global variables of* a module M is the union of

- the set of global variables that are declared in M; and
- the set of all that global variables declared in other modules such that the variables *could* be read or written by a series of procedure calls beginning with a call of one of M's procedures. By "could" we mean the read/write analysis assumes the execution of both branches of conditionals, the execution of while loop bodies, and the terminal of while loops.

To print the global variables of a module M, one runs:

```
print glob M.
```

For example, suppose we make these declarations:

```
module Y1 = {
  var y, z : int
  proc f() : unit = { y <- 0; }
  proc g() : unit = { }
}.
module Y2 = {
  var y : int
  proc f() : unit = { Y1.f(); }
}.
module Y3 = {
  var y : int
```

```
  proc f() : unit = { Y1.g(); }
}.
module type X = {
  proc f() : unit
}.
module Z(X : X) = {
  var y : int
  proc f() : unit = { X.f(); }
}.
```

Then: the set of global variables of `Y1` consists of `Y1.y` and `Y1.z`; the set of global variables of `Y2` consists of `Y1.y` and `Y2.y`; the set of global variables of `Y3` consists of `Y3.y`; the set of global variables of `Z` consists of `Z.y`; and the set of global variables of `Z(Y1)` consists of `Z.y` and `Y1.y`. In the case of `Z`, because its parameter `X` is abstract, no global variables are obtained from `X`.

For every module M, there is a corresponding type, **glob** M, where a value of type **glob** M is a tuple consisting of a value for each of the global variables of M. Nothing can be done with values of such types other than compare them for equality.

2.5 Logics

2.5.1 Formulas

The *formulas* of EasyCrypt's ambient logic are formed by adding to EasyCrypt's expressions

- universal and existential quantification,
- application of built-in and user-defined predicates,
- probability expressions and lossless assertions, and
- HL, pHL and pRHL judgments,

and identifying the formulas with the extended expressions of type `bool`. This means we automatically have all boolean operators as operators on formulas, with their normal precedences and associativities, including negation

```
op [!] : bool -> bool.
```

the two semantically equivalent disjunctions

```
op (||) : bool -> bool -> bool.
op (\/) : bool -> bool -> bool.
```

the two semantically equivalent conjunctions

```
op (&&) : bool -> bool -> bool.
op (/\) : bool -> bool -> bool.
```

implication

```
op (=>) : bool -> bool -> bool.
```

and if-and-only-if

```
op (<=>) : bool -> bool -> bool.
```

The quantifiers' bound identifiers are typed, although EasyCrypt will attempt to infer their types if they are omitted. Universal and existential quantification are written as

```
forall (x : t), φ
```

and

```
exists (x : t), φ
```

respectively, where the formula ϕ typically involves the identifier x of type t. We can abbreviate nested universal or existential quantification in the style of nested anonymous functions, writing, e.g.,

```
forall (x : int, y : int, z : bool), ...
forall (x y : int, z : bool), ...
forall (x y : int) (z : bool), ...
exists (x : int, y : int, z : bool), ...
exists (x y : int, z : bool), ...
exists (x y : int) (z : bool), ...
```

Quantification extends as far to the right as possible, i.e., has lower precedence than the binary operations on formulas.

Abstract *predicates* may be defined as in:

```
pred P0.
pred P1 : int.
pred P2 : int & (int * bool).
pred P3 : int & (int * bool) & (real -> int).
```

`P0`, `P1`, `P2` and `P3` are extended expressions of types `bool`, `int -> bool`, `int -> int * bool -> bool` and `int -> int * bool -> (real -> int) -> bool`, respectively. The parentheses are mandatory in `(int * bool)` and `(real -> int)`. Thus, if e_1, e_2 and e_3 are extended expressions of types `int`, `int * bool` and `real -> int`, respectively, then `P0`, `P1` e_1, `P2` e_1 e_2 and `P3` e_1 e_2 e_3 are formulas.

Concrete predicates are defined in a way that is similar to how operators are declared. E.g., if we declare

```
pred Q (x y : int, z : bool) = x = y /\ z.
```

or

```
pred Q (x : int) (y : int) (z : bool) = x = y /\ z.
```

then `Q` is an extended expression of type

```
int -> int -> bool -> bool
```

meaning that, e.g.,

```
(fun (b : bool -> bool) => b true) (Q 3 4)
```

is a formula. And here is how polymorphic predicates may be defined:

```
pred R ['a, 'b] : ('a, 'a * 'b).
pred R' ['a, 'b] (x : 'a, y : 'a * 'b) = (y.`1 = x).
```

Extended expressions also include program memories, although there isn't a type of memories, and anonymous functions and operators can't take memories as inputs. If `&`m is a memory and x is a program variable that's in `&`m's domain, then `x{`m`}` is the extended expression for the value of x in `&`m. Quantification over memories is allowed:

```
forall &m, φ
```

Here, `&`m ranges over all memories with domains equal to the set of all variables declared as global in currently declared modules. E.g., suppose we have declared:

```
module X = { var x : int }.
module Y = { var y : int }.
```

Then, this is a (true) formula:

```
forall &m, X.x{m} < Y.y{m} => X.x{m} + 1 <= Y.y{m}
```

EasyCrypt's logics can introduce memories whose domains include not just the global variables of modules but also:

- the local variables and parameters of procedures; and
- `res`, whose value in a memory resulting from running a procedure will record the result (return value) of the procedure.

There is no way for the user to introduce such memories directly. We can't do anything with memories other than look up the values of variables in them. In particular, formulas can't test or assert the equality of memories.

If M is a module and $\&m$ is a memory, then $(\texttt{glob}\,M)\{m\}$ is the value of type $\texttt{glob}\,M$ consisting of the tuple whose components are the values of all the global variables of M in $\&m$. (See Subsection 2.4.3 for the definition of the set of all global variables of a module.)

For convenience, we have the following derived syntax for formulas: If ϕ is a formula and $\&m$ is a memory, then $\phi\{m\}$ means the formula in which every subterm u of ϕ consisting of a variable or `res` or $\texttt{glob}\,M$, for a module M, is replaced by $u\{m\}$. For example,

```
(Y1.y = Y1.z => Y1.z = Y1.y){m}
```

expands to

```
Y1.y{m} = Y1.z{m} => Y1.z{m} = Y1.y{m}
```

The parentheses are necessary, because $\_\{m\}$ has higher precedence than even function application. We say that $\&m$ satisfies ϕ iff $\phi\{m\}$ holds.

Extended expressions also include modules, although there isn't a type of modules, and anonymous functions and operators can't take modules as inputs. Quantification over modules is allowed. If T is a module type, and M is a module name, then

$$\texttt{forall}\ (M\ \texttt{<:}\ T),\ \phi$$

means

for all modules M satisfying T, ϕ holds.

Formulas can't talk about module equality.

There is also a variant form of module quantification of the the form

$$\texttt{forall}\ (M\ \texttt{<:}\ T\{N_1,\ldots,N_l\}),\ \phi$$

where $N_1, \ldots, N_l$ are modules, for $l \geq 1$. Its meaning is

for all modules M satisfying T whose sets of global variables
are disjoint from the sets of global variables of the N_i, ϕ holds.

Finally, EasyCrypt's ambient logic has probability expressions, HL, pHL and pRHL judgments, and lossless assertions:

- (**Probability Expressions**) A *probability expression* has the form

  $$\texttt{Pr}[M.p(e_1,\ldots,e_n)\ \texttt{@}\ \&m\ :\phi]$$

  where:

  - p is a procedure of module M that takes n arguments, whose types agree with the types of the e_i;
  - $\&m$ is a memory whose domain is the global variables of all declared modules;

  - the formula ϕ may involve the term `res`, whose type is $M.p$'s return type, as well as global variables of modules.

  Occurrences in ϕ of bound identifiers (bound outside the probability expression) whose names conflict with parameters and local variables of $M.p$ will refer to the bound identifiers, not the parameters/local variables.

  The informal meaning of the probability expression is the probability that running $M.p$ with arguments $e_1, \ldots, e_n$, and initial memory `&`m will terminate in a final memory satisfying ϕ. To run $M.p$:

  - `&`m is extended to map $M.p$'s parameters to the e_i, and to map the procedure's local variables to *arbitrary* initial values;
  - the body of the procedure is run in this extended memory;
  - if the procedure returns, its return value will be stored in a component `res` of the resulting memory, and the procedure's parameters and local variables will be removed from that memory.

  If the procedure doesn't initialize its local variables before using them, the probability expression may be undefined.

- (HL Judgments) A HL *judgment* has the form

  `hoare`[$M.p$: ϕ `==>` ψ]

  where:

  - p is a procedure of module M;
  - the formula ϕ may involve the global variables of declared modules, as well as the parameters of $M.p$;
  - the formula ψ may involve the term `res`, whose type is $M.p$'s return type, as well as the global variables of declared modules.

  Occurrences in ϕ and ψ of bound identifiers (bound outside the judgment) whose names conflict with parameters and local variables of $M.p$ will refer to the bound identifiers, not the parameters/local variables.

  The informal meaning of the HL judgment is that, for all initial memories `&`m satisfying ϕ and whose domains consist of the global variables of declared modules plus the parameters and local variables of $M.p$, if running the body of $M.p$ in `&`m results in termination with a memory, the restriction of that memory to `res` and the global variables of declared modules satisfies ψ.

- (pHL Judgments) A pHL *judgment* has one of the forms

  `phoare` [$M.p$: ϕ `==>` ψ] `<` e
  `phoare` [$M.p$: ϕ `==>` ψ] `=` e
  `phoare` [$M.p$: ϕ `==>` ψ] `>` e

  where:

  - p is a procedure of module M;
  - the formula ϕ may involve the global variables of declared modules, as well as the parameters of $M.p$;
  - the formula ψ may involve the term `res`, whose type is $M.p$'s return type, as well as the global variables of declared modules;
  - e is an expression of type `real`.

Occurrences in ϕ and ψ and of bound identifiers (bound outside the judgment) whose names conflict with parameters or local variables of $M.p$ will refer to the bound identifiers, not the parameters/local variables. `e` will have to be parenthesized unless it is a constant or nullary operator.

The informal meaning of the pHL judgment is that, for all initial memories `&`m satisfying ϕ and whose domains consist of the global variables of declared modules plus the parameters and local variables of $M.p$, the probability that

running the body of $M.p$ in `&`m results in termination
with a memory whose restriction to **`res`** and the global
variables of declared modules satisfies ψ

has the indicated relation to the value of e.

- (pRHL Judgments) A pRHL *judgment* has the form

  `equiv``[`$M.p$ `~` $N.q$ `:` ϕ `==>` ψ`]`

  where:

  - p is a procedure of module M, and q is a procedure of module N;
  - the formula ϕ may involve the global variables of declared modules, the parameters of $M.p$, which must be interpreted in memory `&1` (e.g., `x{1}`), and the parameters of $N.q$, which must be interpreted in memory `&2`;
  - the formula ψ may involve the global variables of declared modules, **`res`**`{1}`, which has the type of $M.p$'s return type, and **`res`**`{2}`, which has the type of $N.q$'s return type.

  Occurrences in ψ of bound identifiers (bound outside the judgment) whose names conflict with parameters and local variables of $M.p$ and $N.q$ will refer to the bound identifiers, not the parameters and local variables, even if they are enclosed in memory references (e.g., `x{1}`). If `&1` (resp., `&2`) is a bound memory (outside the judgment), then all references to `&1` (resp., `&2`) in ϕ and ψ are renamed to use a fresh memory.

  The informal meaning of the pRHL judgment is that, for all initial memories `&1` whose domains consist of the global variables of declared modules plus the parameters and local variables of $M.p$, for all initial memories `&2` whose domains consist of the global variables of declared modules plus the parameters and local variables of $N.q$, if ϕ holds, then the sub-distributions on memories Π_p and Π_q obtained by running $M.p$ on `&1`, storing p's result in the component **`res`** of the resulting memory, from which p's parameters and local variables are removed, and running $N.q$ on `&2`, storing q's result in the component **`res`** of the resulting memory, from which q's parameters and local variables are removed, satisfy ψ, in the following sense. (The probability of a memory in Π_p (resp., Π_q) is the probability that p (resp., q) will terminate with that memory. Π_p and Π_q are *sub*-distributions on memories because p and q may fail to terminate.)

  We say that (Π_p, Π_q) *satisfy* ψ iff there is a function f dividing the probability assigned to each memory `&`m by Π_p among the memories `&`n related to it by ψ (`&`m and `&`n are related according to ψ iff ψ holds when references to `&1` are replaced by reference to `&`m, and reference to `&2` are replaced by reference to `&`n) such that, for all memories `&`n, the value assigned to `&`n by Π_q is the sum of all the probabilities distributed to `&`n by f. (When ψ is an equivalence like `={`**`res`**`}` (i.e., **`res`**`{1} = `**`res`**`{2}`), this is particularly easy to interpret.)

- (Lossless Assertions) A *lossless assertion* has the form

  `islossless` $M.p$

  and is simply an abbreviation for

```
phoare [M.p : true ==> true] = 1%r
```

For the purpose of giving some examples, consider these declarations:

```
module G1 = {
  proc f() : bool = {
    var x : bool;
    x <$ {0,1};
    return x;
  }
}.

module G2 = {
  proc f() : bool = {
    var x, y : bool;
    x <$ {0,1}; y <$ {0,1};
    return x ^^ y;  (* ^^ is exclusive or *)
  }
}.
```

Then:

- The expression

  ```
  Pr[G1.f() @ &m : res]
  ```

  is the probability that `G1.f()` returns `true` when run in the memory `&m`. (The memory is irrelevant, and the expression's value is `1%r / 2%r`.)

- The HL judgement

  ```
  hoare[G2.f : true ==> !res]
  ```

  says that, if `G2.f()` halts (which we know it will), then its return value will be `false`. (This judgement is `false`.)

- The pHL judgement

  ```
  phoare [G2.f : true ==> res] = (1%r / 2%r)
  ```

  says that the probability of `G2.f()` returning `true` is `1%r / 2%r`. (This judgement is `true`.)

- The pRHL judgement

  ```
  equiv[G1.f ~ G2.f : true ==> res]
  ```

  says that `G1.f()` and `G2.f()` are equally likely to return `true` as well as equally likely to return `false`. (This judgement is `true`.)

- The lossless assertion

  ```
  lossless G2.f
  ```

  says that `G2.f()` always terminates, no matter what memory it's run in. (This judgement is `true`.)

2.5.2 Axioms and Lemmas

One states an *axiom* or *lemma* by giving a well-typed formula with no free identifiers, as in:

```
axiom Sym : forall (x y : int), x = y => y = x.
lemma Sym : forall (x y : int), x = y => y = x.
```

The difference between axioms and lemmas is that axioms are trusted by EasyCrypt, whereas lemmas must be proved, in the steps that follow. The *proof* of a lemma has the form

```
proof.
tactic1. ... tacticn.
qed.
```

Actually the `proof` step is optional, but it's good style to include it. The steps of the proof consist of tactic applications; but `print` and `search` commands are also legal steps. The `qed` step saves the lemma, making it available for reuse; it's only allowed when the proof is complete. If the name chosen for a lemma conflicts with an already stated axiom or lemma, one only finds this out upon running `qed`, which will fail. When the proof for a lemma has a very simple form, the proof may be included as part of the lemma's statement:

```
lemma name : φ by [tactic].
```

or

```
lemma name : φ by [].
```

In the first case, the proof consists of a single tactic; the meaning of `by []` will be described in Chapter 3.

One may also parameterize an axiom of lemma by the free identifiers of its formula, as in:

```
lemma Sym (x : int) (y : int) : x = y => y = x.
```

or

```
lemma Sym (x y : int) : x = y => y = x.
```

This version of `Sym` has the same logical meaning as the previous one. But we'll see in Chapter 3 why the parameterized form makes an axiom or lemma easier to apply.

Polymorphic axioms and lemmas may be stated using a syntax reminiscent of the one for polymorphic operators:

```
lemma Sym ['a] (x y : 'a) : x = y => y = x.
lemma PairEq ['a, 'b] (x x' : 'a) (y y' : 'b) :
  x = x' => y = y' => (x, y) = (x', y').
```

or

```
lemma Sym (x y : 'a) : x = y => y = x.
lemma PairEq (x x' : 'a) (y y' : 'b) :
  x = x' => y = y' => (x, y) = (x', y').
```

We can axiomatize the meaning of abstract types, operators and relations. E.g., an abstract type of monoids may be axiomatized by:

```
type monoid.
op id : monoid.
op (+) : monoid -> monoid -> monoid.
axiom LeftIdentity (x : monoid) : id + x = x.
axiom RightIdentity (x : monoid) : x + id = x.
axiom Associative (x y z : monoid) : x + (y + z) = (x + y) + z.
```

Any proofs we do involving monoids will then apply to any valid instantiation of `monoid`, `id` and `(+)`. In Chapter 4, we'll see how to carry out such instantiations using theory cloning.

One must be careful with axioms, however, because it's easy to introduce inconsistencies, allowing one to prove false formulas. E.g., because all types must be nonempty in EasyCrypt, writing

```
type t.
axiom Empty : !(exists (x : t), true).
```

will allow us to prove `false`.

Axioms and lemmas may be parameterized by memories and modules. Consider the declarations:

```
module type T = {
  proc f() : unit
}.
module G(X : T) = {
  var x : int
  proc g() : unit = {
    X.f();
  }
}.
```

Then lemma `Lossless`

```
lemma Lossless (X <: T) : islossless X.f => islossless G(X).g.
```

which is parameterized by an abstract module `X` of module type `T`, says that `G(X).g` always terminates, no matter the memory it's run in, as long as this is true of `X.f`. Lemma `Invar`

```
lemma Invar (X <: T{G}) (n : int) :
  islossless X.f =>
  phoare [G(X).g : G.x = n ==> G.x = n] = 1%r.
```

which is parameterized by an abstract module `X` of module type `T` that is guaranteed not to access or modify `G.x`, and an integer `n`, says that, assuming `X.f` is lossless, if `G(X).g()` is run in a memory giving `G.x` the value `n`, then `G(X).g()` is guaranteed to terminate in a memory in which `G.x`'s value is still `n`. Finally lemma `Invar'`

```
lemma Invar' (X <: T{G}) (n : int) &m :
  islossless X.f => G.x{m} = n =>
  Pr[G(X).g() @ &m : G.x = n] = 1%r.
```

which has the parameters of `Invar` plus a memory `&m`, says the same thing as `Invar`, but using a probability expression rather than a pHL judgement.

Chapter 3

Tactics

Proofs in EasyCrypt are carried out using *tactics*, logical rules embodying general reasoning principles, which transform the current lemma (or *goal*) into zero or more *subgoals*—sufficient conditions for the lemma/goal to hold. Simple ambient logic goals may be automatically proved using SMT solvers.

In this chapter, we introduce EasyCrypt's proof engine, before describing the tactics for EasyCrypt's four logics: ambient, pRHL, pHL and HL.

3.1 Proof Engine

EasyCrypt's proof engine works with goal lists, where a *goal* has two parts:

- A *context* consisting of a
  - a set of type variables, and
  - an *ordered* set of *assumptions*, consisting of identifiers with their types, memories, module names with their module types and restrictions, local definitions, and *hypotheses*, i.e., formulas. An identifier's type may involve the type variables, the local definitions and formulas may involve the type variables, identifiers, memories and module names.
- A *conclusion*, consisting of a single formula, with the same constraints as the assumption formulas.

Informally, to prove a goal, one must show the conclusion to be true, given the truth of the hypotheses, for all valid instantiations of the assumption identifiers, memories and module names.

For example,

```
Type variables: 'a, 'b

x : 'a
x': 'a
y : 'b
y': 'b
eq_xx': x = x'
eq_yy': y = y'
------------------------------------------------------------------------
(x, y) = (x', y')
```

is a goal. And, in the context of the declarations

```
module type T = {
  proc f() : unit
}.
module G(X : T) = {
```

```
  var x : int
  proc g() : unit = {
    X.f();
  }
}.
```

this is a goal:

```
Type variables: <none>

X : T{G}
n : int
LL: islossless X.f
------------------------------------------------------------------------
pre = G.x = n

    G(X).g
    [=] 1%r

post = G.x = n
```

The conclusion of this goal is just a nonlinear rendering of the formula

```
phoare [G(X).g : G.x = n ==> G.x = n] = 1%r.
```

EasyCrypt's pretty printer renders pRHL, pHL and HL judgements in such a nonlinear style when the judgements appear as (as opposed to in) the conclusions of goals.

Internally, EasyCrypt's proof engine also works with pRHL, pHL and HL judgments involving lists of statements rather than procedure names, which we'll call *statement judgements*, below. For example, given this declaration

```
module M = {
  proc f(y : int) = {
    if (y %% 3 = 1) y <- y + 4;
    else y <- y + 2;
    return y;
  }
}.
```

this is an pHL statement judgement:

```
Type variables: <none>

x : int
zor1_x: x = 1 \/ x = 2
------------------------------------------------------------------------
Context : M.f

pre = y %% 3 = x

(1--)  if (y %% 3 = 1) {
(1.1)    y <- y + 4
(1--)  } else {
(1?1)    y <- y + 2
(1--)  }

post = y %% 3 = x %% 2 + 1
```

The pre- and post-conditions of a statement judgement may refer to the parameters and local variables of the *procedure context* of the conclusion—`M.f` in the preceding example. They may also refer to the memories `&1` and `&2` in the case of pRHL statement judgements. When a statement judgement appears anywhere other than as the conclusion of a goal, the pretty printer renders it in abbreviated linear syntax. E.g., the preceding goal is rendered as

```
hoare[if (x %% 3 = 1) {···} : x %% 3 = n ==> x %% 3 = n %% 2 + 1]
```

Statement judgements can't be directly input by the user.

We use the term *program* to refer to *either* a procedure appearing in a pRHL, pHL or HL judgement, *or* a statement list appearing in a pRHL, pHL or HL statement judgement. In the case of pRHL (statement) judgements, we speak of the *left* and *right* programs, also using *program 1* for the left program, and *program 2* for the right one. We will only speak of a program's *length* when it's a statement list we are referring to. By the *empty* program, we mean the statement list with no statements.

When the proof of a lemma is begun, the proof engine starts out with a single goal, consisting of the lemma's statement. E.g., the lemma

```
lemma PairEq ['a, 'b] :
  forall (x x' : 'a) (y y' : 'b),
  x = x' => y = y' => (x, y) = (x', y').
```

gives rise to the goal

```
Type variables: 'a, 'b

------------------------------------------------------------------------
forall (x x' : 'a) (y y' : 'b), x = x' => y = y' => (x, y) = (x', y')
```

For parameterized lemmas, the goal includes the lemma's parameters as assumptions. E.g.,

```
lemma PairEq (x x' : 'a) (y y' : 'b) :
  x = x' => y = y' => (x, y) = (x', y').
```

gives rise to

```
Type variables: 'a, 'b

x : 'a
x': 'a
y : 'b
y': 'b
------------------------------------------------------------------------
x = x' => y = y' => (x, y) = (x', y')
```

EasyCrypt's tactics, when applicable, reduce the first goal to zero or more subgoals. E.g., if the first goal is

```
Type variables: <none>

x : int
zor1_x: x = 1 \/ x = 2
------------------------------------------------------------------------
Context : M.f

pre = y %% 3 = x

(1--)  if (y %% 3 = 1) {
(1.1)    y <- y + 4
(1--)  } else {
(1?1)    y <- y + 2
(1--)  }

post = y %% 3 = x %% 2 + 1
```

then applying the **if** tactic (handle a conditional) reduces (replaces) this goal with the two goals

```
Type variables: <none>

x : int
zor1_x: x = 1 \/ x = 2
------------------------------------------------------------------------
Context : M.f

pre = y %% 3 = x /\ y %% 3 = 1

(1)  y <- y + 4

post = y %% 3 = x %% 2 + 1
```

and

```
Type variables: <none>

x : int
zor1_x: x = 1 \/ x = 2
------------------------------------------------------------------------
Context : M.f

pre = y %% 3 = x /\ y %% 3 <> 1

(1)  y <- y + 2

post = y %% 3 = x %% 2 + 1
```

(leaving the remaining goals, if any, unchanged). If the first goal is

```
Type variables: <none>

x : int
zor1_x: x = 1 \/ x = 2
------------------------------------------------------------------------
forall &hr,
  y{hr} %% 3 = x /\ y{hr} %% 3 = 1 => (y{hr} + 4) %% 3 = x %% 2 + 1
```

then applying the **smt** tactic (try to solve the goal using SMT provers) solves the goal, i.e., replaces it with no subgoals. Applying a tactic may fail; in this case an error message is issued and the list of goals is left unchanged.

A lemma's proof may be saved, using the step **qed**, when the list of goals becomes empty. And this must be done before anything else may be done.

Remark. In the descriptions of EasyCrypt's tactics given in the following two sections, unless otherwise specified, you should assume that the subgoals to which a tactic reduces a goal have the same contexts as that original goal.

3.2 Matching

Statement patterns are an extension of statements. **FiXme Note: Add explanation of new `replace` tactic, which uses statement patterns.**

Blocks	
{}	empty statment
{s}	start and end anchors around s
{s]	start anchor at the beginning of s
[s}	end anchor at the end of s
[s]	find s at any position in the statments, without looking into possible branches
<s>	find s at any position in the statments, and looking into possible branches

Statments	
_	any sequence of statments
n!_	a sequence of n statments
s as X	s that is associated with the name X
X	interpreted as : _ as X
!s	repeat s, can be zero time
?s	s is to appear once, or zero time
n!s	repeat s n times
[n..m]!s	repeat s at least n times, up to m times
[n..]!s	repeat s at least n times
[..m]!s	repeat s up to m times
~s	apply not greedy carateristics to s if that makes sense
s1 ; s2	s1 followed directly by s2
s1 s2	s1 followed directly by s2
s1 | s2	s1 or s2
_ <- _ ;	any affectation
_ <$ _ ;	any sample
_ <@ _ ;	any procedure call
if ;	any if statment
while ;	any while statment
if _ bt else bf	a if statment where bt is the block matched in the branch true's corpse, and bf the block in the branch false's body
while _ b	a while branch where the block b is matched in the corpse of the loop

3.3 Ambient logic

In this section, we describe the proof terms, tactics and tacticals of EasyCrypt's ambient logic.

3.3.1 Proof Terms

Formulas introduce identifier and formula assumptions using universal quantifiers and implications. For example, the formula

```
forall (x y : bool), x = y => forall (z : bool), y = z => x = z.
```

introduces the assumptions

```
x     : bool
y     : bool
eq_xy : x = y
z     : bool
eq_yz : y = z
```

(where the names of the two formulas were chosen to be meaningful), and has `x = z` as its conclusion. We refer to the first assumption of a formula as the formula's *top assumption*. E.g., the top assumption of the preceding formula is `x : bool`.

EasyCrypt has *proof terms*, which partially describe how to prove a formula. Their syntax is described in Figure 3.1, where X ranges over lemma (or formula assumption) names. A proof term for a lemma (or formula assumption) X has components corresponding to the assumptions introduced by X. A component corresponding to a variable consists of an expression of the

p	::=	_	**proof hole**
		$(X, q_1, \ldots, q_n)$	**lemma application**
q	::=	e	**expression**
		p	**proof term**

Figure 3.1: Proof Terms

variable's type. The proof term is explaining how the instantiation of the lemma's conclusion with these expressions may be proved. A formula component consists of a proof term explaining how the instantiation of the formula may be proved. Proof holes will get turned into subgoals when a proof term is used in backward reasoning, e.g., by the `apply` (p. 60) tactic. **FiXme Note: Need explanation of how a proof term may be used in forward reasoning.**

Consider, e.g., the following declarations and axioms

```
pred P : int.
pred Q : int.
pred R : int.
axiom P (x : int) : P x.
axiom Q (x : int) : P x => Q x.
axiom R (x : int) : P(x + 1) => Q x => R x.
```

Then, given that `x : int` is an assumption,

```
(R x (P(x + 1)) (Q x (P x)))
```

is a proof term proving the conclusion `R x`. And

```
(R x _ (Q x _))
```

is a proof term that turns proofs of `P(x + 1)` and `P x` into proofs of `R x`. When used in backward reasoning, it will reduce a goal with conclusion `R x` to subgoals with conclusions `P(x + 1)` and `P x`. **FiXme Note: Can it be used in forward reasoning?**

Some of a proof term's expressions may be replaced by `_`, asking EasyCrypt to infer them from the context. Going even further, one may abbreviate a one-level proof term all of whose components are `_` to just its lemma name. For example, we can write `R` for `(R _ _ _)`. When used in backward reasoning, it will reduce a goal with conclusion `R x` to subgoals with conclusions `P(x + 1)` and `Q x`. **FiXme Note: In forward reasoning they aren't equivalent—why?**

3.3.2 Occurrence Selectors and Rewriting Directions

Some ambient logic tactics use *occurrence selectors* to restrict their operation to certain occurrences of a term or formula in a goal's conclusion or formula assumption. The syntax is $\{i_1, \ldots, i_n\}$, specifying that only occurrences i_1 throught i_n of the term/formula in a depth-first, left-to-right traversal of the goal's conclusion or formula assumption should be operated on. Specifying $\{-i_1, \ldots, i_n\}$ restricts attention to all occurrences *not* in the following list. They may also be empty, meaning that all applicable occurrences should be operated on.

Some ambient logic tactics use *rewriting directions*, *dir*, which may either be empty (meaning rewriting from left to right), or `-`, meaning rewriting from right to left.

3.3.3 Introduction and Generalization

Introduction. One moves the assumptions of a goal's conclusion into the goal's context using the introduction tactical. This tactical uses introduction patterns, which are defined in Figure 3.2. In this definition, *occ* ranges over occurrence selectors, and *dir* ranges over directions—see Subsection 3.3.2).

If a list $\iota_1, \ldots, \iota_n$ of introduction patterns consists entirely of `//` (apply the `trivial` (p. 55) tactic), `/=` (apply the `simplify` (p. 56) tactic) and `//=` (apply the `simplify` and then `trivial`),

ι	::=	b	**name**
		`_`	**no name**
		`+`	**auto revert**
		`?`	**find name**
		occ `->`	**rewrite using assumption**
		occ `<-`	**rewrite in reverse using assumption**
		`->>`	**substitute using assumption**
		`<<-`	**substitute in reverse using assumption**
		`/`*p*	**replace assumption by applying proof term**
		`{`$a_1 \cdots a_n$`}`	**clear introduced assumptions**
		`/=`	**simplify**
		`//`	**trivial**
		`//=`	**simplify then trivial**
		dir occ `@/`*op*	**unfold definition of operator**
		`[`$\iota_{11} \cdots \iota_{1m_1}$ `|` $\cdots$ `|` $\iota_{r1} \cdots \iota_{rm_r}$`]`	**case pattern**
b	::=	x	**identifier**
		M	**module name**
		`&`m	**memory name**

Figure 3.2: Introduction Patterns

then *applying* $\iota_1, \ldots, \iota_n$ *to* a list of goals $G_1, \ldots, G_m$ is done by applying the tactics corresponding to the ι_i in order to each G_j, causing some of the goals to be solved and thus disappear and some of the goals to be simplified.

τ`=>` $\iota_1 \cdots \iota_n$

Runs the tactic τ, matching the resulting goals, $G_1, \ldots, G_l$, with the introduction patterns $\iota_1, \ldots, \iota_n$:

- Suppose k is such that all of $\iota_1, \ldots, \iota_{k-1}$ are `//`, `/=` and `//=`, and either $k > n$ or ι_k is not `//`, `/=` or `//=`.
- Let $G'_1, \ldots, G'_{l'}$ be the goals resulting from applying $\iota_1, \ldots, \iota_{k-1}$ to $G_1, \ldots, G_l$.
- If $l' = 0$, the tactical produces no subgoals.
- Otherwise, if $k > n$, the tactical's result is $G'_1, \ldots, G'_{l'}$.
- Otherwise, if ι_k is not a case pattern, each subgoal G'_i is matched against $\iota_k, \ldots, \iota_n$ by the procedure described below, with the resulting subgoals being collected into a list of goals (maintaining order viz a viz the indices i) as the tactical's result.
- Otherwise, ι_k is a case pattern `[`$\iota_{11} \cdots \iota_{1m_1}$ `|` $\cdots$ `|` $\iota_{r1} \cdots \iota_{rm_r}$`]`.
- If τ is not equivalent to `idtac` (p. 47), the tactic fails unless $r = l'$, in which case each G'_i is matched against

  $$\iota_{i1} \cdots \iota_{im_i} \iota_{k+1} \cdots \iota_n$$

  by the procedure described below, with the resulting subgoals being collected into the tactical's result.
- Otherwise, τ is equivalent to `idtac` (p. 47) (and so $l' = 1$). In this case G'_1 is matched against $\iota_k, \ldots, \iota_n$ by the procedure described below, with the resulting subgoals being collected into a list of goals as the tactical's result.

Matching a single goal against a list of patterns: To match a goal G against a list of introduction patterns $\iota_1, \ldots, \iota_n$, the introduction patterns are processed from left-to-right, as follows:

- (b) The top assumption (universally quantified identifier, module name or memory; or left side of implication) is consumed, and introduced with this name. Fails if the top assumption has neither of these forms.
- (b `!`) Same as the preceding case, except that b is used as the base of the introduced name, extending the base to avoid naming conflicts.
- (`_`) Same as the preceding case, except the assumption is introduced with an anonymous name (which can't be uttered by the user).
- (`+`) Same as the preceding case, except that after a branch of the procedure completes, yielding a goal, the assumption will be reverted, i.e., un-introduced (using a universal quantifier or implication as appropriate).
- (`?`) Same as the preceding case, except EasyCrypt chooses the name by which the assumption is introduced (using universally quantified names as assumption bases).
- (occ `->`) Consume the top assumption, which must be an equality, and use it as a left-to-right rewriting rule in the remainder of the goal's conclusion, restricting rewriting to the specified occurrences of the equality's left side.
- (occ `<-`) The same as the preceding case, except the rewriting is from right-to-left.
- (`->>`) The same as `->`, except the consumed equality assuption is used to perform a left-to-right substitution in the entire goal, i.e., in its assumptions, as well as its conclusion.
- (`<<-`) The same as the preceding case, except the substitution is from right-to-left.
- (`/`p) Replace the top assumption by the result of applying the proof term p to it using forward reasoning.
- (`{`$a_1 \cdots a_n$`}`) Doesn't affect the goal's conclusion, but clears the specified assumptions, i.e., removes them. Fails if one or more of the assumptions can't be cleared, because a remaining assumption depends upon it.
- (`/=`) Apply **`simplify`** (p. 56) to goal's conclusion.
- (`//`) Apply **`trivial`** (p. 55) to goal's conclusion; this may solve the goal, i.e., so that the procedure's current branch yields no goals.
- (`/=`) Apply **`simplify`** (p. 56) and then **`trivial`** (p. 55) to goal's conclusion; this may solve the goal, so that the procedure's current branch yields no goals.
- (dir occ `@/`op) Unfold (fold, if the direction is `-`) the definition of operator op at the specified occurrences of the goal's conclusion. See the **`rewrite`** (p. 62) tactic for the details.
- (`[`$\iota_{11} \cdots \iota_{1m_1}$ `|` $\cdots$ `|` $\iota_{r1} \cdots \iota_{rm_r}$`]`)
  - If $r = 0$, then the top assumption of the goal is destructed using the **`case`** (p. 64) tactic, the resulting goals are matched against $\iota_2, \ldots, \iota_n$, and their subgoals are assembled into a list of goals.
  - Otherwise $r > 0$. The goal's top assumption is destructed using the **`case`** (p. 64) tactic, yielding subgoals $H_1, \ldots H_p$. If $p \neq r$, the procedure fails. Otherwise each subgoal H_i is matched against

  $$\iota_{i1} \cdots \iota_{im_i} \iota_2 \cdots \iota_n$$

  with the resulting goals being collected into a list as the procedure's result.

The following examples use the tactic `move` (p. 47), which is equivalent to `idtac` (p. 47). In its simplest form, the introduction tactical simply gives names to assumptions. For example, if the current goal is

```
Type variables: <none>

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

then running

```
move=> x y eq_xy z eq_yz.
```

produces

```
Type variables: <none>

x : int
y : int
eq_xy: x = y
z : int
eq_yz: y = z

x = z
```

Alternatively, we can use the introduction pattern `?` to let EasyCrypt choose the assumption names, using `H` as a base for formula assumptions and starting from the identifier names given in universal quantifiers:

```
move=> ? ? ? ? ?.
```

produces

```
Type variables: <none>

x : int
y : int
H : x = y
z : int
H0: y = z

x = z
```

To see how the `->` rewriting pattern works, suppose the current goal is

```
Type variables: <none>

x : int
y : int

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

Then running

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

produces

```
Type variables: <none>

x : int
y : int

forall (z : int), y = z => y = z
```

Alternatively, one can introduce the assumption `x = y`, and then use the ->> substitution pattern: if the current goal is

```
Type variables: <none>

x : int
y : int
H1: x = y
z : int
------------------------------------------------------------
y = z => x = z
```

then running

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

produces

```
Type variables: <none>

x : int
y : int
z : int
H1: x = z
------------------------------------------------------------
x = z
```

To see how a view may be applied to a not-yet-introduced formula assumption, suppose the current goal is

```
Type variables: <none>

Sym: forall (u v : int), u = v => v = u
x : int
y : int
------------------------------------------------------------
x = y => forall (z : int), y = z => x = z
```

Then running

```
move=> /Sym.
```

produces

```
Type variables: <none>

Sym: forall (u v : int), u = v => v = u
x : int
y : int
------------------------------------------------------------
y = x => forall (z : int), y = z => x = z
```

And then running

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

on this goal produces

```
Type variables: <none>

Sym: forall (u v : int), u = v => v = u
x : int
y : int
------------------------------------------------------------
forall (z : int), x = z => x = z
```

Finally, let's see examples of how a disjunction assumption may be destructed, either using the `case` tactic followed by a case introduction pattern, or by making the case introduction pattern do the destruction. For the first case, if the current goal is

```
Type variables: <none>

x : bool
y : bool
________________________________________
x \/ y =>
(x => forall (z : bool), P x z) =>
(y => forall (z w : bool), Q y z w) =>
(forall (z : bool), P x z) \/ forall (z w : bool), Q y z w
```

then running

```
move=> [Hx HP _ | Hy _ HQ].
```

produces the two goals

```
Type variables: <none>

x : bool
y : bool
Hx: x
HP: x => forall (z : bool), P x z
_ : y => forall (z w : bool), Q y z w
________________________________________
(forall (z : bool), P x z) \/ forall (z w : bool), Q y z w
```

and

```
Type variables: <none>

x : bool
y : bool
Hy: y
_ : x => forall (z : bool), P x z
HQ: y => forall (z w : bool), Q y z w
________________________________________
(forall (z : bool), P x z) \/ forall (z w : bool), Q y z w
```

And for the second case, if the current goal is

```
Type variables: <none>

x : bool
y : bool
________________________________________
x \/ y =>
(x => forall (z : bool), P x z) =>
(y => forall (z w : bool), Q y z w) =>
(forall (z : bool), P x z) \/ forall (z w : bool), Q y z w
```

then running

```
case=> [Hx HP X | Hy X HQ] {X}.
```

produces the two goals

```
Type variables: <none>

x : bool
```

```
y : bool
Hx: x
HP: x => forall (z : bool), P x z
------------------------------------------------------------
(forall (z : bool), P x z) \/ forall (z w : bool), Q y z w
```

and

```
Type variables: <none>

x : bool
y : bool
Hy: y
HQ: y => forall (z w : bool), Q y z w
------------------------------------------------------------
(forall (z : bool), P x z) \/ forall (z w : bool), Q y z w
```

Note how we used the clear pattern to discard the assumption X.

Generalization. The generalization tactical moves assumptions from the context into the conclusion and generalizes subterms or formulas of the conclusion.

τ: $\pi_1 \cdots \pi_n$

Generalize the patterns $\pi_1, \ldots, \pi_n$, starting from π_n and going back, and then run tactic τ. *This tactical is only applicable to certain tactics:* **move** *(p. 47),* **case** *(p. 64) (just the version that destructs the top assumption of a goal's conclusion) and* **elim** *(p. 67).*

- When π is an assumption from the context, it's moved back into the conclusion, using universal quantification or an implication, as appropriate. If one assumption depends on another, one can't generalize the later without also generalizing the former.

  For example, if the current goal is

  ```
  Type variables: <none>

  x : int
  y : int
  eq_xy: x = y
  ------------------------------------------------------------
  y = x
  ```

  then running

  ```
  move: x eq_xy.
  ```

  produces

  ```
  Type variables: <none>

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

  In this example, one can't generalize x without also generalizing eq_xy.

- π may also be a subformula or subterm of the goal, or _, which stands for the whole goal, possibly prefixed by an occurrence selector. This replaces the formula or subterm with a universally quantified identifier of the approprate type.

  For example, if the current goal is

```
Type variables: <none>

x : int
y : int
--------------------------------
x = y => y = x
```

then running

```
move: (y = x).
```

produces

```
Type variables: <none>

x : int
y : int
--------------------------------
forall (x0 : bool), x = y => x0
```

Alternatively, running

```
move: {2} y {2} x.
```

produces

```
Type variables: <none>

x : int
y : int
--------------------------------
forall (y0 x0 : int), x = y => y0 = x0
```

3.3.4 Tactics

`idtac`

Does nothing, i.e., leaves the goal unchanged.

`move`

Does nothing, equivalent to `idtac` (p. 47). It is mainly used in conjunction with the introduction tactical and the generalization mechanism. See Section 3.3.3.

`clear` $a_1 \cdots a_n$

Clear assumptions $a_1 \cdots a_n$ from the goal's context. Fail if any remaining hypotheses depend on any of the a_i.
For example, if the current goal is

```
Type variables: <none>

x : int
y : int
z : int
eq_xy: x = y
eq_yz: y = z
--------------------------------
x - 1 = y - 1
```

then running

```
clear z eq_yz.
```

produces

```
Type variables: <none>

x : int
y : int
eq_xy: x = y
--------------------------------------------------------------
x - 1 = y - 1
```

assumption

Search in the context for a hypothesis that is convertible to the goal's conclusion, solving the goal if one is found. Fail if none can be found.
For example, if the current goal is

```
Type variables: <none>

x : int
y : int
eq_xy: x = y
--------------------------------------------------------------
x = y
```

then running

```
assumption.
```

solves the goal.

reflexivity

Solve goals with conclusions of the form `b = b` (up to computation).
For example, if the current goal is

```
Type variables: <none>

y : bool
--------------------------------------------------------------
(fun (x : bool) => !x) y = !y
```

then running

```
reflexivity.
```

solves the goal.

left

Reduce a goal whose conclusion is a disjunction to one whose conclusion is its left member.
For example, if the current goal is

```
Type variables: <none>

x : int
y : int
eq_xy: x - 1 = y
--------------------------------------------------------------
x = y + 1 \/ y = x + 1
```

then running

```
left.
```

produces the goal

```
Type variables: <none>

x : int
y : int
eq_xy: x - 1 = y
________________________________________
x = y + 1
```

right

Reduce a goal whose conclusion is a disjunction to one whose conclusion is its right member. For example, if the current goal is

```
Type variables: <none>

x : int
y : int
eq_yz: y - 1 = x
________________________________________
x = y + 1 \/ y = x + 1
```

then running

```
right.
```

produces the goal

```
Type variables: <none>

x : int
y : int
eq_yz: y - 1 = x
________________________________________
y = x + 1
```

If we replace `\/` by `||` in this example, we can see the difference between the two versions of disjunction: if the current goal is

```
Type variables: <none>

x : int
y : int
eq_yz: y - 1 = x
________________________________________
x = y + 1 || y = x + 1
```

then running

```
right.
```

produces the goal

```
Type variables: <none>

x : int
y : int
eq_yz: y - 1 = x
________________________________________
x <> y + 1 => y = x + 1
```

exists e

Reduces proving an existential to proving the witness e satisfies the existential's body. For example, if the current goal is

```
Type variables: <none>

x : int
rng_x: 0 < x < 5
------------------------------------------------------------
exists (x0 : int), 5 < x0 < 10
```

then running

```
exists (x + 5).
```

produces the goal

```
Type variables: <none>

x : int
rng_x: 0 < x < 5
------------------------------------------------------------
5 < x + 5 < 10
```

split

Break a goal whose conclusion is intrinsically conjunctive into goals whose conclusions are its conjuncts. For instance, it can:

- close any goal that is convertible to `true` or provable by **reflexivity**,
- replace a logical equivalence by the direct and indirect implications,
- replace a goal of the form ϕ_1 `/\` ϕ_2 by the two subgoals for ϕ_1 and ϕ_2. The same applies for a goal of the form ϕ_1 `&&` ϕ_2,
- replace an equality between n-tuples by n equalities on their components.

For example, if the current goal is

```
Type variables: <none>

x : int
y : int
------------------------------------------------------------
x = y <=> x - 1 = y - 1
```

then running

```
split.
```

produces the goals

```
Type variables: <none>

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

and

```
Type variables: <none>

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

And if the current goal is

```
Type variables: <none>

x : int
y : int
z : int
w : int
eq_xy: x = y
eq_zw: z = w
----------------------------------------------------------------------
x - 1 = y - 1 /\ z + 1 = w + 1
```

then running

```
split.
```

produces the goals

```
Type variables: <none>

x : int
y : int
z : int
w : int
eq_xy: x = y
eq_zw: z = w
----------------------------------------------------------------------
x - 1 = y - 1
```

and

```
Type variables: <none>

x : int
y : int
z : int
w : int
eq_xy: x = y
eq_zw: z = w
----------------------------------------------------------------------
z + 1 = w + 1
```

Repeating the last example with `&&` rather than `/\`, if the current goal is

```
Type variables: <none>

x : int
y : int
z : int
w : int
eq_xy: x = y
eq_zw: z = w
----------------------------------------------------------------------
x - 1 = y - 1 && z + 1 = w + 1
```

then running

```
split.
```

produces the goals

```
Type variables: <none>

x : int
y : int
z : int
w : int
eq_xy: x = y
eq_zw: z = w
-----------------------------------------------------------------
x - 1 = y - 1
```

and

```
Type variables: <none>

x : int
y : int
z : int
w : int
eq_xy: x = y
eq_zw: z = w
-----------------------------------------------------------------
x - 1 = y - 1 => z + 1 = w + 1
```

This illustrates the difference between `/\` and `&&`. And if the current goal is

```
Type variables: <none>

x : int
y : int
z : int
w : int
eq_xz: x = z + 9
eq_yw: y = w - 12
-----------------------------------------------------------------
(x - 7, 2 + y) = (z + 2, w - 10)
```

then running

```
split.
```

produces the goals

```
Type variables: <none>

x : int
y : int
z : int
w : int
eq_xz: x = z + 9
eq_yw: y = w - 12
-----------------------------------------------------------------
x - 7 = z + 2
```

and

```
Type variables: <none>

x : int
y : int
z : int
w : int
eq_xz: x = z + 9
eq_yw: y = w - 12
---------------------------------
2 + y = w - 10
```

congr

Replace a goal whose conclusion has the form $f\,t_1\,\cdots\,t_n = f\,u_1\,\cdots\,u_n$, where f is an assumption identifier or operator, with subgoals having conclusions $t_i = u_i$ for all i. Subgoals solvable by **reflexivity** are automatically closed. Also works when the operator is used in infix form.
For example, if the current goal is

```
Type variables: <none>

x : int
y : int
x': int
y': int
f : int -> int -> int
eq_xx': x = x' - 2
eq_yy': y = y' + 2
---------------------------------
f (x + 1) (y - 1) = f (x' - 1) (y' + 1)
```

then running

```
congr.
```

produces the goals

```
Type variables: <none>

x : int
y : int
x': int
y': int
f : int -> int -> int
eq_xx': x = x' - 2
eq_yy': y = y' + 2
---------------------------------
x + 1 = x' - 1
```

and

```
Type variables: <none>

x : int
y : int
x': int
y': int
f : int -> int -> int
eq_xx': x = x' - 2
eq_yy': y = y' + 2
---------------------------------
y - 1 = y' + 1
```

And if the current goal is

```
Type variables: <none>

x : int
y : int
x': int
y': int
eq_xx': x = x' - 2
eq_yy': y = y' + 2
------------------------------------------------------------------------
x + 1 + (y - 1) = x' - 1 + (y' + 1)
```

then running

```
congr.
```

produces this same pair of subgoals.

subst x | **subst**

Syntax: `subst` x. Search for the first equation of the form $x = t$ or $t = x$ in the context and replace all the occurrences of x by t everywhere in the context and the conclusion before clearing it.
For example, if the current goal is

```
Type variables: <none>

x : bool
y : bool
z : bool
w : bool
eq_yx: y = x
eq_yz: y = z
eq_zw: z = w
------------------------------------------------------------------------
x = w
```

then running

```
subst x.
```

takes us to

```
Type variables: <none>

y : bool
z : bool
w : bool
eq_zw: z = w
eq_yz: y = z
------------------------------------------------------------------------
y = w
```

from which running

```
subst y.
```

takes us to

```
Type variables: <none>

z : bool
w : bool
```

```
eq_zw: z = w
─────────────────────────────
z = w
```

from which running

```
subst z.
```

takes us to

```
Type variables: <none>

w : bool
─────────────────────────────
w = w
```

Syntax: `subst`. Repeatedly apply `subst` x to all identifiers in the context.
For example, if the current goal is

```
Type variables: <none>

x : bool
y : bool
z : bool
w : bool
eq_yx: y = x
eq_yz: y = z
eq_zw: z = w
─────────────────────────────
x = w
```

then running

```
subst.
```

takes us to

```
Type variables: <none>

x : bool
─────────────────────────────
x = x
```

`trivial`

Try to solve the goal by using a mixture of low-level tactics. This tactic is called by the introduction pattern //.
For example, if the current goal is

```
Type variables: <none>

─────────────────────────────
forall (x y : int), x = y => y - 1 = x - 1
```

then running

```
trivial.
```

solves the goal. On the other hand, if the current goal is

```
Type variables: <none>

─────────────────────────────
forall (x y z : bool), (x /\ y => z) <=> (x => y => z)
```

then running

```
trivial.
```

leaves the goal unchanged.

FiXme Note: Be a bit more detailed about what this tactic does?

`done`

Apply `trivial` (p. 55) and fail if the goal is not closed.

`simplify` | `simplify` $x_1 \cdots x_n$ | `simplify delta`

Reduce the goal's conclusion to its $\beta\iota\zeta\Lambda$-head normal-form, followed by one step of parallel, strong δ-reduction if `delta` is given. The δ-reduction can be restricted to a set of defined symbols by replacing `delta` by a non-empty sequence of targeted symbols. You can reduce the conclusion to its β-head normal form (resp. ι, ζ, Λ-head normal form) by using the tactic `beta` (resp. `iota`, `zeta`, `logic`). These tactics can be combined together, separated by spaces, to perform head reduction by any combination of the rule sets.
For example, if the current goal is

```
Type variables: <none>

y : bool
z : bool
imp_yz: y => z
----------------------------------------------------------------------
true => true <=> (y => true /\ z)
```

then running

```
simplify.
```

produces the goal

```
Type variables: <none>

y : bool
z : bool
imp_yz: y => z
----------------------------------------------------------------------
y => z
```

And if the current goal is

```
Type variables: <none>

y : bool
z : bool
imp_yz: y => z
----------------------------------------------------------------------
false => false <=> (y => false /\ z)
```

then running

```
simplify.
```

produces the goal

```
Type variables: <none>

y : bool
z : bool
```

```
imp_yz: y => z
------------------------------------------------
true
```

FiXme Note: Is this the right place to define "convertible"?

◎ `progress` | `progress` τ

Break the goal into multiple *simpler* ones by repeatedly applying `split`, `subst` and `move=>`. The tactic τ given to `progress` is tentatively applied after each step.
For example, if the current goal is

```
Type variables: <none>

------------------------------------------------
forall (x y z : bool), (x /\ y => z) <=> (x => y => z)
```

then running

```
progress.
```

solves the goal.

FiXme Note: Describe `progress` options.

◎ `smt` *smt-options*

Try to solve the goal using SMT solvers. The goal is sent along with the local hypotheses plus selected axioms and lemmas. The SMT solvers used, their options, and the axiom selection algorithm are specified by *smt-option*.
For example, if the current goal is

```
Type variables: <none>

------------------------------------------------
forall (x y z : bool), (x /\ y => z) <=> (x => y => z)
```

then running

```
smt.
```

solves the goal.

Options

- `timeout`=n: set the timeout for provers to n (in seconds).
- `maxprovers`=n: set the maximun number of prover runing in parallele to n
- `prover`=[*prover-selector*]: select the provers, where *prover-selector* is a list of modified prover names:
  - ``` `` ```*prover-name*`''`: use the listed prover;
  - `+`` `*prover-name*`''`: add *prover-name* to the current list of provers;
  - `-`` `*prover-name*`''`: remove *prover-name* from the current list of provers;

  Examples:
  - `[``Z3'' ``Alt-Ergo'']`: use only Z3 and Alt-Ergo;
  - `[``Z3'' ``Alt-Ergo'' -''Z3'']`: use only Alt-Ergo;
  - `[-''CVC4'']`: remove CVC4 form the current list of provers;
  - `[+''CVC4'']`: add CVC4 to the current list of provers;

  FiXme Note: Describe failure states of prover selection.
- Axiom selection: axioms and lemmas are not all send to smt provers, EasyCrypt use a strategy to automatically select them. Lemmas and axioms marked with "nosmt" are not selected by default. This strategy can be parametrized using different options:
  - `unwantedlemmas=`*dbhint*: do not send axiom/lemma selected by *dbhint*
  - `wantedlemmas=`*dbhint*: send axiom/lemma selected by *dbhint*

  - `all`: select all available axioms/lemmas execpted those specified by `unwantedlemmas` (if any).
  - `maxlemmas=`n: set the maximun number of selected axioms/lemmas to n. Keep this number small is generally more effienciant. Variant: n
  - `iterate`: try to incrementally augment the number of selected axioms/lemmas. Last call will be equivalent to all.

FiXme Note: Describe *dbhint* options.

Variant: Short options.
Options can also be specified by short name, for example:

```
smt 100 [+''Z3] tmo=4 mp=2
```

is equivalent to

```
smt maxlemmas=100 prover=[+''Z3] timeout=4 maxprovers=2
```

Short options can be any substring of the full option names that uniquely identifies the desired option: when several options match, their full names are given.
Smt option can be set globally using the following syntax: `prover` *smt-options* **FiXme Note: Make this a pragma?**
Remark: By default, `smt` failures cannot be caught by the `try` (p. 69) tactical.

◎ `admit`

Close the current goal by admitting it.
For example, if the current goal is

```
Type variables: <none>

x : int
y : int
--------------------------------------------
x = y + 1 /\ x = y - 1
```

then running

```
admit.
```

solves the goal.

◎ `change` ϕ

Replace the current goal's conclusion by ϕ—ϕ must be *convertible* to the current goal's conclusion.
For example, if the current goal is

```
Type variables: <none>

y : bool
z : bool
imp_yz: y => z
--------------------------------------------
true => true <=> (y => true /\ z)
```

then running

```
change (y => z).
```

produces the goal

```
Type variables: <none>

y : bool
z : bool
```

```
imp_yz: y => z
----------------------------------------
y => z
```

pose x := π

Search for the first subterm t of the goal's conclusion matching π and leading to the full instantiation of the pattern. Then introduce to the goal's context, after instantiation, the local definition x := t, and abstract all occurrences of t in the goal's conclusion as x. An occurence selector can be used (see Subsection 3.3.2).
For example, if the current goal is

```
Type variables: <none>

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

then running

```
pose z := (_ + y) * _.
```

produces the goal

```
Type variables: <none>

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

have ι: ϕ

Logical cut. Generate two subgoals: one whose conclusion is the cut formula ϕ, and one with conclusion ϕ => ψ where ψ is the current goal's conclusion. Moreover, the introduction pattern ι is applied to the second subgoal.
For example, if the current goal is

```
Type variables: <none>

x : bool
notnot_x: (x => false) => false
----------------------------------------
x
```

then running

```
cut excl_or : x \/ (x => false).
```

produces the goals

```
Type variables: <none>

x : bool
notnot_x: (x => false) => false
----------------------------------------
x \/ (x => false)
```

and

```
Type variables: <none>

x : bool
notnot_x: (x => false) => false
excl_or: x \/ (x => false)
--------------------------------------------------
x
```

Syntax: `have` ι: ϕ `by` τ. Attempts to use tactic τ to close the first subgoal (corresponding to the cut formula ϕ), and fails if impossible.

⊚ `cut` ι: ϕ

Same as `have` (p. 59).

⊚ `apply` p | `apply` $/p_1 \cdots /p_n$ | `apply` p `in` H

Syntax: `apply` p. Tries to match the conclusion of the proof term p with the goal's conclusion. If the match succeeds and leads to the full instantiation of the pattern, then the goal is replaced, after instantiation, with the subgoals of the proof term.
Consider the declarations

```
pred P : int.
pred Q : int.
pred R : int.
axiom P (x : int) : P x.
axiom Q (x : int) : P x => Q x.
axiom R (x : int) : P(x + 1) => Q x => R x.
```

If the current goal is

```
Type variables: <none>

x : int
--------------------------------------------------
R x
```

then running

```
apply R.
```

produces the goals

```
Type variables: <none>

x : int
--------------------------------------------------
P (x + 1)
```

and

```
Type variables: <none>

x : int
--------------------------------------------------
Q x
```

And running

```
apply (R x _ (Q x _)).
```

from that initial goal produces the goals

```
Type variables: <none>

x : int
--------------------------------------------------------------------
P (x + 1)
```

and

```
Type variables: <none>

x : int
--------------------------------------------------------------------
P x
```

Syntax: `apply` $/p_1 \cdots /p_n$. Apply the proof terms $p_1, \ldots, p_n$ in sequence. At each stage of this process, we have some number of goals. Initially, we have just the current goal. After applying p_1, we have whatever goals p_1 has produced from the current goal. p_2 is applied to the *last* of these goals, and that last goal is replaced by the goals produced by running p_2, etc. Fails without changing the goal if any of these applications fails.
For example, if the current goal is

```
Type variables: <none>

x : int
--------------------------------------------------------------------
R x
```

then running

```
apply /R /Q /P /P.
```

solves the goal.

Syntax: `apply` p `in` H. Apply p in forward reasoning to H, replacing H by the result.
For example, if the current goal is

```
Type variables: <none>

x : int
HP: P (x + 1)
--------------------------------------------------------------------
Q x => R x
```

then running

```
apply R in HP.
```

produces the goal

```
Type variables: <none>

x : int
HP: Q x => R x
--------------------------------------------------------------------
Q x => R x
```

◎ `exact` p | `exact` $/p_1 \cdots /p_n$

Syntax: `exact` p. Equivalent to `by apply` p, i.e., apply the given proof-term and then try to close the goals with `trivial`—failing if not all goals can be closed.

Syntax: `exact` $/p_1 \cdots /p_n$. Equivalent to `by apply` $/p_1 \cdots /p_n$.

◎ `rewrite` $\pi_1 \cdots \pi_n$ | `rewrite` $\pi_1 \cdots \pi_n$ `in` H

Syntax: `rewrite` $\pi_1 \cdots \pi_n$. Rewrite the rewrite-pattern $\pi_1 \cdots \pi_n$ from left to right, where the π_i can be of the following form:

- one of `//`, `/=`, `//=`,
- a proof-term p, or
- a pattern prefixed by `/` (slash).

The two last forms can be prefixed by a direction indicator (the sign `-`, see Subsection 3.3.2), followed by an occurrence selector (see Subsection 3.3.2), followed (for proof-terms only) by a repetition marker (`!`, `?`, `n!` or `n?`). All these prefixes are optional.

Depending on the form of π, `rewrite` π does the following:

- For `//`, `/=`, and `//=`, see Subsection 3.3.3.
- If π is a proof-term with conclusion $f_1 = f_2$, then `rewrite` searches for the first subterm of the goal's conclusion matching f_1 and resulting in the full instantiation of the pattern. It then replaces, after instantiation of the pattern, all the occurrences of f_1 by f_2 in the goal's conclusion, and creates new subgoals for the instantiations of the assuptions of p. If no subterms of the goal's conclusion match f_1 or if the pattern cannot be fully instantiated by matching, the tactic fails. The tactic works the same if the pattern ends by f_1 `<=>` f_2. If the direction indicator `-` is given, `rewrite` works in the reverse direction, searching for a match of f_2 and then replacing all occurrences of f_2 by f_1.
- If π is a `/`-prefixed pattern of the form $o\, p_1 \cdots p_n$, with o a defined symbol, then `rewrite` searches for the first subterm of the goal's conclusion matching $o\, p_1 \cdots p_n$ and resulting in the full instantiation of the pattern. It then replaces, after instantiation of the pattern, all the occurrences of $o\, p_1 \cdots p_n$ by the $\beta\delta$ head-normal form of $o\, p_1 \cdots p_n$, where the δ-reduction is restricted to subterms headed by the symbol o. If no subterms of the goal's conclusion match $o\, p_1 \cdots p_n$ or if the pattern cannot be fully instantiated by matching, the tactic fails. If the direction indicator `-` is given, `rewrite` works in the reverse direction, searching for a match of the $\beta\delta_o$ head-normal of $o\, p_1 \cdots p_n$ and then replacing all occurrences of this head-normal form with $o\, p_1 \cdots p_n$.

The occurrence selector restricts which occurrences of the matching pattern are replaced in the goal's conclusion—see Subsection 3.3.2.
Repetition markers allow the repetition of the same rewriting. For instance, `rewrite` π leads to `do! rewrite` π. See the tactical `do` for more information.
Lastly, `rewrite` $\pi_1 \cdots \pi_n$ is equivalent to `rewrite` π_1; ...; `rewrite` π_n.

For example, if the current goal is

```
Type variables: <none>

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

then running

```
rewrite eq_xy.
```

produces

```
Type variables: <none>

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

from which running

```
rewrite - {1} eq_xy.
```

produces

```
Type variables: <none>

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

from which running

```
rewrite - eq_xy.
```

produces

```
Type variables: <none>

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

Syntax: `rewrite` $\pi_1 \cdots \pi_n$ `in` H. Like the preceding case, except rewriting is done in the hypothesis H instead of in the goal's conclusion. Rewriting using a proof term is only allowed when the proof term was defined globally or before the assumption H.
For example, if the current goal is

```
Type variables: <none>

x : int
y : int
eq_xy: x = y
z : int
eq_xz: y = z
------------------------------------------------------------
x = z
```

then running

```
rewrite - eq_xy in eq_xz.
```

produces

```
Type variables: <none>

x : int
y : int
eq_xy: x = y
```

```
z : int
eq_xz: x = z
________________________________________
x = z
```

case ϕ | **case**

Syntax: `case` ϕ. Assuming the goal's conclusion is *not* a statement judgement, do an excluded-middle case analysis on ϕ, substituting ϕ in the goal's conclusion.
For example, if the current goal is

```
Type variables: <none>

x : int
y : int
abs_bnd: `|x - y| <= 10
________________________________________
x - y <= 10 \/ y - x <= 10
```

then running

```
case (x <= y).
```

produces the goals

```
Type variables: <none>

x : int
y : int
abs_bnd: `|x - y| <= 10
________________________________________
x <= y => x - y <= 10 \/ y - x <= 10
```

and

```
Type variables: <none>

x : int
y : int
abs_bnd: `|x - y| <= 10
________________________________________
! x <= y => x - y <= 10 \/ y - x <= 10
```

Syntax: `case`. Destruct the top assumption of the goal's conclusion, generating subgoals that are dependent upon the kind of assumption destructed. *This form of the tactic can be followed by the generalization tactical—see Subsection 3.3.3.*

- (**conjunction**) For example, if the current goal is

  ```
  Type variables: <none>

  x : int
  y : int
  z : int
  ________________________________________
  x = y /\ y = z => x = z
  ```

  then running

  ```
  case.
  ```

produces the goal

```
Type variables: <none>

x : int
y : int
z : int
----------------------------------------
x = y => y = z => x = z
```

`&&` works identically.

- (**disjunction**) For example, if the current goal is

```
Type variables: <none>

x : int
y : int
----------------------------------------
x < y \/ y < x => x - y <> 0
```

then running

```
case.
```

produces the goals

```
Type variables: <none>

x : int
y : int
----------------------------------------
x < y => x - y <> 0
```

and

```
Type variables: <none>

x : int
y : int
----------------------------------------
y < x => x - y <> 0
```

`||` works identically.

- (**existential**) For example, if the current goal is

```
Type variables: <none>

----------------------------------------
(exists (x : int), 0 < x < 5) => exists (x : int), 5 < x < 10
```

then running

```
case.
```

produces the goal

```
Type variables: <none>

----------------------------------------
forall (x : int), 0 < x < 5 => exists (x0 : int), 5 < x0 < 10
```

- (**unit**) Substitutes `tt` for the assumption in the remainder of the conclusion.
- (**bool**) For example, if the current goal is

```
Type variables: <none>

forall (x y : bool), x <=> y => x = y
```

then running

```
case.
```

produces the goals

```
Type variables: <none>

forall (y : bool), true <=> y => true = y
```

and

```
Type variables: <none>

forall (y : bool), false <=> y => false = y
```

- (**product type**) For example, if the current goal is

```
Type variables: <none>

forall (x y : bool * bool),
  x <> y => x.`1 <> y.`1 \/ x.`2 <> y.`2
```

then running

```
case.
```

produces the goal

```
Type variables: <none>

forall (x1 x2 : bool) (y : bool * bool),
  (x1, x2) <> y =>
  (x1, x2).`1 <> y.`1 \/ (x1, x2).`2 <> y.`2
```

- (**inductive datatype**) Consider the inductive datatype declaration:

```
type tree = [Leaf | Node of bool & tree & tree].
```

Then, if the current goal is

```
Type variables: <none>

forall (tr : tree),
  tr = Leaf \/
  exists (b : bool) (tr1 tr2 : tree), tr = Node b tr1 tr2
```

then running

```
case.
```

produces the goals

```
Type variables: <none>

________________________________________________
Leaf = Leaf \/
exists (b : bool) (tr1 tr2 : tree), Leaf = Node b tr1 tr2
```

and

```
Type variables: <none>

________________________________________________
forall (b : bool) (t t0 : tree),
  Node b t t0 = Leaf \/
  exists (b0 : bool) (tr1 tr2 : tree),
    Node b t t0 = Node b0 tr1 tr2
```

`elim` | `elim` */L*

Syntax: `elim`. Eliminates the top assumption of the goal's conclusion, generating subgoals that are dependent upon the kind of assumption eliminated. *This tactic can be followed by the generalization tactical—see Subsection 3.3.3.*
`elim` mostly works identically to `case` (p. 64), the exception being inductive datatype and the integers (for which a built-in induction principle is applied—see the other form).
Consider the inductive datatype declaration:

```
type tree = [Leaf | Node of bool & tree & tree].
```

Then, if the current goal is

```
Type variables: <none>

p : tree -> bool
basis: p Leaf
indstep: forall (b : bool) (tr1 tr2 : tree),
           p tr1 => p tr2 => p (Node b tr1 tr2)
________________________________________________
forall (tr : tree), p tr
```

running

```
elim.
```

produces the goals

```
Type variables: <none>

p : tree -> bool
basis: p Leaf
indstep: forall (b : bool) (tr1 tr2 : tree),
           p tr1 => p tr2 => p (Node b tr1 tr2)
________________________________________________
p Leaf
```

and

```
Type variables: <none>

p : tree -> bool
basis: p Leaf
indstep: forall (b : bool) (tr1 tr2 : tree),
           p tr1 => p tr2 => p (Node b tr1 tr2)
------------------------------------------------------------
forall (b : bool) (t t0 : tree), p t => p t0 => p (Node b t t0)
```

Syntax: `elim` /*L*. Eliminates the top assumption of the goal's conclusion using the supplied induction principle lemma. *This tactic can be followed by the generalization tactical—see Subsection 3.3.3*. For example, consider the declarations

```
type tree = [Leaf | Node of bool & tree & tree].
op rev (tr : tree) : tree =
  with tr = Leaf => Leaf
  with tr = Node b tr1 tr2 => Node b (rev tr1) (rev tr2).
```

and suppose we've already proved

```
lemma IndPrin :
  forall (p : tree -> bool) (tr : tree),
  p Leaf =>
  (forall (b : bool) (tr1 tr2 : tree),
   p tr1 => p tr2 => p(Node b tr1 tr2)) =>
  p tr.
```

Then, if the current goal is

```
Type variables: <none>

------------------------------------------------------------
forall (t : tree), rev (rev t) = t
```

running

```
elim /IndPrin.
```

produces the goals

```
Type variables: <none>

------------------------------------------------------------
rev (rev Leaf) = Leaf
```

and

```
Type variables: <none>

------------------------------------------------------------
forall (b : bool) (tr1 tr2 : tree),
  rev (rev tr1) = tr1 =>
  rev (rev tr2) = tr2 => rev (rev (Node b tr1 tr2)) = Node b tr1 tr2
```

When we consider the `Int` theory in Chapter 5, we'll discuss the induction principle on the integers.

`algebra`

FiXme Note: Missing description of `algebra`.

3.3.5 Tacticals

Tactics can be combined together, composed and modified by *tacticals*. We've already seen the introduction and generalization tacticals, which turn a tactic τ and a list of patterns into a composite tactic, which may then combined with other tactics.

⊚ τ_1 ; τ_2

Apply τ_2 to all the subgoals generated by τ_1. Sequencing groups to the left, so that τ_1 ; τ_2 ; τ_3 means (τ_1 ; τ_2) ; τ_3.
For example, if the current goal is

```
Type variables: <none>

x : bool
y : bool
z : bool
--------------------------------------------------
(x /\ y) /\ z => x /\ y /\ z
```

then running

```
case; case.
```

produces the goals

```
Type variables: <none>

x : bool
y : bool
z : bool
--------------------------------------------------
x => y => z => x /\ y /\ z
```

⊚ τ_1 ; [τ_1 | ··· | τ_n]

Run τ_1, which must generate exactly n subgoals, $G_1, \ldots, G_n$. Then apply τ_i' to G_i, for all i.
For example, if the current goal is

```
Type variables: <none>

x : bool
y : bool
z : bool
tr_x: x
tr_y: y
tr_z: z
--------------------------------------------------
x /\ y /\ z
```

then running

```
split; [assumption | split; assumption].
```

solves the goal.

⊚ **try** τ

Execute the tactic τ if it succeeds; do nothing (leave the goal unchanged) if it fails.

Remark. By default, EasyCrypt proofs are run in `strict` mode. In this mode, `smt` failures cannot be caught using `try`. This allows EasyCrypt to always build the proof tree correctly, even in weak check mode, where `smt` calls are assumed to succeed. Inside a strict proof, weak check mode can be turned on and off at will, allowing for the fast replay of proof sections during development. In any event, we recommend *never* using `try smt`: a little thought is much more cost-effective than failing `smt` calls.

`do!` τ

Apply τ to the current goal, then repeatedly apply it to all subgoals, stopping on a branch only when it fails. An error is produced it τ does not apply to the current goal.

For example, if the current goal is

```
Type variables: <none>

x : bool
y : bool
z : bool
w : bool
------------------------------------------------------------------------
((x /\ y) /\ z) /\ w \/ (x /\ y /\ z) /\ w => w
```

then running

```
do! case.
```

produces the goals

```
Type variables: <none>

x : bool
y : bool
z : bool
w : bool
------------------------------------------------------------------------
x => y => z => w => w
```

and

```
Type variables: <none>

x : bool
y : bool
z : bool
w : bool
------------------------------------------------------------------------
x => y /\ z => w => w
```

Variants.

`do?` τ	apply τ 0 or more times, until it fails
`do` n`!` τ	apply τ with depth exactly n
`do` n`?` τ	apply τ with depth at most n

τ; `first` τ_2

Apply the tactic τ_1, then apply τ_2 on the first subgoal generated by `t1`, leaving the other goals unchanged. An error is produced if no subgoals are generated by τ_1.

Variants.

τ_1 ; `first` n τ_2	apply τ_2 on the first n subgoals generated by τ_1
τ_1 ; `last` τ_2	apply τ_2 on the last subgoal generated by τ_1
τ_1 ; `last` n τ_2	apply τ_2 on the last n subgoals generated by τ_1
τ; `first` n `last`	reorder the subgoals generated by τ, moving the first n to the end of the list
τ; `last` n `first`	reorder the subgoals generated by τ, moving the last n to the beginning of the list
τ; `last first`	reorder the subgoals generated by τ, moving the last one to the beginning of the list
τ; `first last`	reorder the subgoals generated by τ, moving the first one to the end of the list

For example, if the current goal is

```
Type variables: <none>

x : bool
y : bool
z : bool
------------------------------------------
(x /\ y => z) <=> (x => y => z)
```

then running

```
split; last first.
```

produces the goals

```
Type variables: <none>

x : bool
y : bool
z : bool
------------------------------------------
(x => y => z) => x /\ y => z
```

and

```
Type variables: <none>

x : bool
y : bool
z : bool
------------------------------------------
(x /\ y => z) => x => y => z
```

◎ `by` τ

Apply the tactic τ and try to close all the generated subgoals using `trivial` (p. 55). Fail if not all subgoals can be closed.

Remark. Inside the a lemma's proof, `by []` is equivalent to `by trivial`. But the form

```
lemma ··· by [].
```

means

```
lemma ··· by (trivial; smt).
```

3.4 Program Logics

In this section, we describe the tactics of EasyCrypt's three program logics: pRHL, pHL and HL. There are five rough classes of program logic tactics:

1. those that actually reason about the program in Hoare logic style;
2. those that correspond to semantics-preserving program transformations or compiler optimizations;
3. those that operate at the level of specifications, strenghtening, combining or splitting goals without modifying the program;
4. tactics that automate the application of other tactics;
5. advanced tactics for handling eager/lazy sampling and bounding the probability of failure.

We discuss these five classes in turn.

Some of the program reasoning tactics have two modes when used on goals whose conclusions are pRHL statement judgements. Their default mode is to operate on both programs at once. When a side is specified (using τ{1} or τ{2}), a one-sided variant is used, with 1 referring to the left program, and 2 to the right one.

3.4.1 Tactics for Reasoning about Programs

⊙ `proc`

Syntax: `proc`. Turn a goal whose conclusion is a pRHL, pHL or HL judgement involving *concrete* procedure(s) into one whose conclusion is a statement judgement by replacing the concrete procedure(s) by their body/ies. Assertions about `res`/`res`{i} are turned into ones about the value(s) returned by the procedure(s).

For example, if the current goal is

```
Type variables: <none>

pre = ={x, y}

    G1.f ~ G2.f

post = ={res}
```

then running

```
proc.
```

produces the goal

```
Type variables: <none>

&1 (left ) : G1.f
&2 (right) : G2.f

pre =
  (x{1}, y{1}).`1 = (x{2}, y{2}).`1 /\
  (x{1}, y{1}).`2 = (x{2}, y{2}).`2

x <- x + 1                  (1)  y <- y + 1
y <- y + 1                  (2)  x <- x + 1

post = x{1} + y{1} = x{2} + y{2}
```

Syntax: `proc` I. Reduce a goal whose conclusion is a pRHL judgement involving the same *abstract* procedure (but perhaps using different implementations of its oracles) (resp., an HL judgement involving an abstract proceure) to goals whose conclusions are pRHL (resp., HL) judgements on those oracles, plus goals with ambient logic conclusions checking the original judgement's pre- and postconditions allow such a reduction (the preconditions *must* assume I and, in the pRHL case, the equality of the abstract procedure's parameter(s) and the global variables of the module in which the procedure is contained (except in the case when the module's type specifies that the abstract procedure initializes its global variables; the postconditions *may* assume I and, in the pRHL case, the equality of the results of the procedure call(s) and the values of the global variables). The generated pRHL/HL subgoals have pre- and postconditions assuming/asserting I; in the pRHL case, the preconditions also assume the equality of the oracles' parameters, and their postconditions also assert the equality of the oracles' results).

For example, given the declarations

```
module type OR = {
  proc f1() : unit
  proc f2() : unit
  proc f3() : unit
}.

module Or : OR = {
  var x : int
  proc f1() : unit = {
    x <- x + 2;
  }
  proc f2() : unit = {
    x <- x - 2;
  }
  proc f3() : unit = {
    x <- x + 1;
  }
}.

module type T(O : OR) = {
  proc g() : unit {O.f1 O.f2}
}.
```

if the current goal is

```
Type variables: <none>

M : T{Or}
------------------------------------------------------------------
pre = ={glob M} /\ Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0

    M(Or).g ~ M(Or).g

post = Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0
```

then running

```
proc (Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0).
```

produces the goals

```
Type variables: <none>

M : T{Or}
------------------------------------------------------------------
forall &1 &2,
```

```
  ={glob M} /\ Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0 =>
  true /\ ={glob M} /\ Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0
```

and

```
Type variables: <none>

M : T{Or}
────────────────────────────────────────────────────────────────────
forall &1 &2,
  ={res} /\ ={glob M} /\ Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0 =>
  Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0
```

and

```
Type variables: <none>

M : T{Or}
────────────────────────────────────────────────────────────────────
pre = true /\ Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0

    Or.f1 ~ Or.f1

post = ={res} /\ Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0
```

and

```
Type variables: <none>

M : T{Or}
────────────────────────────────────────────────────────────────────
pre = true /\ Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0

    Or.f2 ~ Or.f2

post = ={res} /\ Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0
```

The tactic would fail without the module restriction `T{Or}` on `M`, as then `M` could directly manipulate `Or.x`. It would also fail if, in the declaration of the module type `T`, `g` were given access to `O.f3`.

Syntax: **proc** B I. Like **proc** I, but just for pRHL judgements and uses "upto-bad" (upto-failure) reasoning, where the *bad* (failure) event, B, is evaluated in the second program's memory, and the invariant I only holds up to the point when failure occurs. In addition to subgoals whose conclusions are pRHL judgments involving the oracles the abstract procedure may query (their preconditions assume I and the equality of oracles' parameters, as well as that B is false; their postconditions assert I and the equality of the oracles' results—but only when B does not hold), subgoals are generated that check that: the original judgement's pre- and postconditions support the reduction; the abstract procedure is lossless, assuming the losslessness of the oracles it may query; the oracles used by the abstract procedure in the first program are lossless once the bad event occurs; and the oracles used by the abstract procedure in the second program guarantee the stability of the failure event with probability 1.

For example, suppose we have the following declarations

```
module type OR = {
  proc qry(x : int) : int
}.

op low : int = -100.
op upp : int = 100.
```

```
module Or1 : OR = {
  var qry, rsp : int
  var queried : bool

  proc qry(x : int) : int = {
    var y : int;
    if (x = qry) {
      y <- rsp;
      queried <- true;
    } else {
      y <$ [low .. upp];
    }
    return y;
  }
}.

module Or2 : OR = {
  var qry : int
  var queried : bool

  proc qry(x : int) : int = {
    var y : int;
    y <$ [low .. upp];
    queried <- queried \/ x = qry;
    return y;
  }
}.

module type ADV(O : OR) = {
  proc * f() : bool
}.
```

Then, if the current goal is

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------------
pre = Or1.qry{1} = Or2.qry{2}

    Adv(Or1).f ~ Adv(Or2).f

post = !Or2.queried{2} => ={res}
```

running

```
proc Or2.queried (Or1.qry{1} = Or2.qry{2}).
```

produces the goals

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------------
forall &1 &2,
  Or1.qry{1} = Or2.qry{2} =>
  !Or2.queried{2} => true /\ Or1.qry{1} = Or2.qry{2}
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------------
forall &1 &2,
  (!Or2.queried{2} =>
   ={res} /\ ={glob Adv} /\ Or1.qry{1} = Or2.qry{2}) =>
  !Or2.queried{2} => ={res}
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------------
forall (O <: OR{Adv}), islossless O.qry => islossless Adv(O).f
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------------
pre = !Or2.queried{2} /\ ={x} /\ Or1.qry{1} = Or2.qry{2}

    Or1.qry ~ Or2.qry

post = !Or2.queried{2} => ={res} /\ Or1.qry{1} = Or2.qry{2}
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------------
forall &2, Or2.queried{2} => islossless Or1.qry
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------------
forall _,
  phoare[ Or2.qry : Or2.queried /\ true ==> Or2.queried /\ true] = 1%r
```

Syntax: `proc` $B\ I\ J$. Like `proc` $B\ I$, but where the extra invariant, J, holds *after* failure has occurred. In the pRHL subgoals involving oracles called by the abstract procedure: the preconditions assume I and the equality of the oracles' parameters, as well as that B is false; and the postconditions assert

- I and the equality of the oracles' results—when B does not hold; and
- J—when B does hold.

For example, given the declarations of the **proc** B I example, if the current goal is

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------------
pre = Or1.qry{1} = Or2.qry{2} /\ Or1.queried{1} = Or2.queried{2}

    Adv(Or1).f ~ Adv(Or2).f

post = Or1.queried{1} = Or2.queried{2} /\ (!Or2.queried{2} => ={res})
```

then running

```
proc Or2.queried
           (Or1.qry{1} = Or2.qry{2} /\ Or1.queried{1} = Or2.queried{2})
           (Or1.queried{1} = Or2.queried{2}).
```

produces the goals

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------------
forall &1 &2,
  Or1.qry{1} = Or2.qry{2} /\ Or1.queried{1} = Or2.queried{2} =>
  if Or2.queried{2} then Or1.queried{1} = Or2.queried{2}
  else
    true /\
    Or1.qry{1} = Or2.qry{2} /\ Or1.queried{1} = Or2.queried{2}
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------------
forall &1 &2,
  if Or2.queried{2} then Or1.queried{1} = Or2.queried{2}
  else
    ={res} /\
    ={glob Adv} /\
    Or1.qry{1} = Or2.qry{2} /\ Or1.queried{1} = Or2.queried{2} =>
  Or1.queried{1} = Or2.queried{2} /\ (!Or2.queried{2} => ={res})
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
```

```
forall (O <: OR{Adv}), islossless O.qry => islossless Adv(O).f
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------------
pre =
  !Or2.queried{2} /\
  ={x} /\ Or1.qry{1} = Or2.qry{2} /\ Or1.queried{1} = Or2.queried{2}

    Or1.qry ~ Or2.qry

post =
  if Or2.queried{2} then Or1.queried{1} = Or2.queried{2}
  else
    ={res} /\
    Or1.qry{1} = Or2.qry{2} /\ Or1.queried{1} = Or2.queried{2}
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------------
forall &2,
  Or2.queried{2} =>
  phoare[ Or1.qry :
           Or1.queried = Or2.queried{2} ==>
           Or1.queried = Or2.queried{2}] = 1%r
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------------
forall &1,
  phoare[ Or2.qry :
           Or2.queried /\ Or1.queried{1} = Or2.queried ==>
           Or2.queried /\ Or1.queried{1} = Or2.queried] = 1%r
```

Syntax: `proc*`. Reduce a pRHL (resp., HL) judgement to a pRHL (resp., HL) statement judgement involving calls (resp., a call) to the procedures (resp., procedure).

For example, if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
pre = ={x, y}

    G1.f ~ G2.f
```

```
post = ={res}
```

then running

```
proc*.
```

produces the goal

```
Type variables: <none>

________________________________________________________________________
&1 (left ) : G1.f
&2 (right) : G2.f

pre = ={x, y}

r <@ G1.f(x, y)            (1)  r <@ G2.f(x, y)

post = ={r}
```

Remark. This tactic is particularly useful in combination with `inline` (p. 106) when faced with a pRHL judgment where one of the procedures is concrete and the other is abstract.

`skip`

If the goal's conclusion is a statement judgement whose program(s) are empty, reduce it to the goal whose conclusion is the ambient logic formula P => Q, where P is the original conclusion's precondition, and Q is its postcondition.

For example, if the current goal is

```
Type variables: <none>

________________________________________________________________________
&1 (left ) : M.f
&2 (right) : N.f

pre = x{1} = -x{2} /\ ={y}


post = x{1} + y{1} = - (x{2} - y{2})
```

then running

```
skip.
```

produces the goal

```
Type variables: <none>

________________________________________________________________________
forall &1 &2, x{1} = -x{2} /\ ={y} => x{1} + y{1} = - (x{2} - y{2})
```

`seq`

Syntax: `seq` n_1 n_2 : R. **pRHL sequence rule.** If n_1 and n_2 are natural numbers and the goal's conclusion is a pRHL statement judgement with precondition P, postcondition Q and such that the lengths of the first and second programs are at least n_1 and n_2, respectively, then reduce the goal to two subgoals:

- A first goal whose conclusion has precondition P, postcondition R, first program consisting of the first n_1 statements of the original goal's first program, and second program consisting of the first n_2 statements of the original goal's second program.
- A second goal whose conclusion has precondition R, postcondition Q, first program consisting of all but the first n_1 statements of the original goal's first program, and second program consisting of all but the first n_2 statements of the original goal's second program.

For example, if the current goal is

```
Type variables: <none>

&1 (left ) : G1.f
&2 (right) : G2.f

pre = ={x, y}

x <- x + 1                (1)  y <- y + 1
y <- y + 1                (2)  x <- x + 1

post = x{1} + y{1} = x{2} + y{2}
```

then running

```
seq 1 1 : (x{1} = x{2} + 1 /\ y{2} = y{1} + 1).
```

produces the goals

```
Type variables: <none>

&1 (left ) : G1.f
&2 (right) : G2.f

pre = ={x, y}

x <- x + 1                (1)  y <- y + 1

post = x{1} = x{2} + 1 /\ y{2} = y{1} + 1
```

and

```
Type variables: <none>

&1 (left ) : G1.f
&2 (right) : G2.f

pre = x{1} = x{2} + 1 /\ y{2} = y{1} + 1

y <- y + 1                (1)  x <- x + 1

post = x{1} + y{1} = x{2} + y{2}
```

Syntax: `seq` n : R. **HL sequence rule.** If n is a natural number and the goal's conclusion is an HL statement judgement with precondition P, postcondition Q and such that the length of the program is at least n, then reduce the goal to two subgoals:

- A first goal whose conclusion has precondition P, postcondition R, and program consisting of the first n statements of the original goal's program.

- A second goal whose conclusion has precondition R, postcondition Q, and program consisting of all but the first n statements of the original goal's program.

For example, if the current goal is

```
Type variables: <none>

Context : M.f

pre = x %% 2 = 0

(1)  x <- x + 1
(2)  x <- x + 1

post = x %% 2 = 0
```

then running

```
seq 1 : (x %% 2 = 1).
```

produces the goals

```
Type variables: <none>

Context : M.f

pre = x %% 2 = 0

(1)  x <- x + 1

post = x %% 2 = 1
```

and

```
Type variables: <none>

Context : M.f

pre = x %% 2 = 1

(1)  x <- x + 1

post = x %% 2 = 0
```

sp

Syntax: `sp`. If the goal's conclusion is a pRHL, pHL or HL statement judgement, consume the longest prefix(es) of the conclusion's program(s) consisting entirely of statements built-up from ordinary assignments (not random assignments or procedure call assignments) and `if` statements, replacing the conclusion's precondition by the strongest postcondition R such that the statement judgement consisting of the conclusion's original precondition, the consumed prefix(es) and R holds.

For example, if the current goal is

```
Type variables: <none>
```

```
&1 (left ) : M.f
&2 (right) : N.f

pre = ={y}

if (y = 0) {               (1--)  if (y = 0) {
  y <- 2                   (1.1)    y <- 3
} else {                   (1--)  } else {
  y <- y - 2               (1?1)    y <- y - 1
}                          (1--)  }
x <@ M.g()                 (2--)  x <$ [1..10]

post = x{1} + y{1} = x{2} + y{2} - 1
```

then running

```
sp.
```

produces the goal

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f

pre =
  (exists (y_R : int),
     y{2} = 3 /\
     y_R = 0 /\
     ((exists (y_L : int), y{1} = 2 /\ y_L = 0 /\ y_L = y_R) \/
      exists (y_L : int), y{1} = y_L - 2 /\ y_L <> 0 /\ y_L = y_R)) \/
  exists (y_R : int),
    y{2} = y_R - 1 /\
    y_R <> 0 /\
    ((exists (y_L : int), y{1} = 2 /\ y_L = 0 /\ y_L = y_R) \/
     exists (y_L : int), y{1} = y_L - 2 /\ y_L <> 0 /\ y_L = y_R)

x <@ M.g()                 (1)  x <$ [1..10]

post = x{1} + y{1} = x{2} + y{2} - 1
```

Syntax: `sp` n_1 n_2. In PRHL, let `sp` consume *exactly* n_1 statements of the first program and n_2 statements of the second program. Fails if this isn't possible.

Syntax: `sp` n. In PHL and HL, let `sp` consume *exactly* n statements of the program. Fails if this isn't possible.

wp

Syntax: `wp`. If the goal's conclusion is a PRHL, PHL or HL statement judgement, consume the longest suffix(es) of the conclusion's program(s) consisting entirely of statements built-up from ordinary assignments (not random assignments or procedure call assignments) and `if` statements, replacing the conclusion's postcondition by the weakest precondition R such that the statement judgement consisting of R, the consumed suffix(es) and the conclusion's original postcondition holds.

For example, if the current goal is

```
Type variables: <none>
```

```
&1 (left ) : M.f
&2 (right) : N.f

pre = ={y}

x <@ M.g()                  (1--)  x <$ [1..10]
if (y = 0) {                (2--)  if (y = 0) {
  y <- 2                    (2.1)    y <- 3
} else {                    (2--)  } else {
  y <- y - 2                (2?1)    y <- y - 1
}                           (2--)  }

post = x{1} + y{1} = x{2} + y{2} - 1
```

then running

```
wp.
```

produces the goal

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f

pre = ={y}

x <@ M.g()                  (1)  x <$ [1..10]

post =
  if y{2} = 0 then
    if y{1} = 0 then x{1} + 2 = x{2} + 3 - 1
    else x{1} + (y{1} - 2) = x{2} + 3 - 1
  else
    let y_R = y{2} - 1 in
    (if y{1} = 0 then x{1} + 2 = x{2} + y_R - 1
     else x{1} + (y{1} - 2) = x{2} + y_R - 1)
```

Syntax: `wp` n_1 n_2. In pRHL, let `wp` consume *exactly* n_1 statements of the first program and n_2 statements of the second program. Fails if this isn't possible.

Syntax: `wp` n. In pHL and HL, let `wp` consume *exactly* n statements of the program. Fails if this isn't possible.

⊙ `rnd`

When describing the variants of this tactic, we ony consider random assignments whose left-hand sides consist of single identifiers. The generalization to multiple assignment, when distributions over tuple types are sampled, is straightforward.

Syntax: `rnd` | `rnd` f | `rnd` f g. If the conclusion is a pRHL statement judgement whose programs end with random assignments x_1 <\$ d_1 and x_2 <\$ d_2, and f and g are functions between the types of x_1 and x_2, then consume those random assignments, replacing the conclusion's postcondition by the probabilistic weakest precondition of the random assignments wrt. f and g. The new postcondition checks that:

- f and g are an isomorphism between the distributions d_1 and d_2;

- for all elements u in the support of d_1, the result of substituting u and $f\,u$ for x_1`{1}` and x_2`{2}` in the conclusion's original postcondition holds.

When g is f, it can be omitted. When f is the identity, it can be omitted.

For example, if the current goal is

```
Type variables: <none>

n : int
------------------------------------------------------------------------
&1 (left ) : M.h
&2 (right) : N.h

pre = y{2} = n

x <$ {0,1}                 (1)  y <- y - 1
                           (2)  x <$ [2..3]

post = x{1} <=> x{2} + y{2} = n + 2
```

then running

```
rnd (fun b => b ? 3 : 2) (fun m => m = 3).
```

produces the goal

```
Type variables: <none>

n : int
------------------------------------------------------------------------
&1 (left ) : M.h
&2 (right) : N.h

pre = y{2} = n

                           (1)  y <- y - 1

post =
  (forall (xR : int),
     xR \in [2..3] => xR = if xR = 3 then 3 else 2) &&
  (forall (xR : int),
     xR \in [2..3] => mu1 [2..3] xR = mu1 {0,1} (xR = 3)) &&
  forall (xL : bool),
    xL \in {0,1} =>
    ((if xL then 3 else 2) \in [2..3]) &&
    xL = ((if xL then 3 else 2) = 3) &&
    (xL <=> (if xL then 3 else 2) + y{2} = n + 2)
```

Note that if one uses the other isomorphism between `{0,1}` and `[2..3]` the generated subgoal will be false.

Syntax: `rnd{1} | rnd{2}`. If the conclusion is a PRHL statement judgement whose designated program (1 or 2) ends with a random assignment x <\$ d, then consume that random assignment, replacing the conclusion's postcondition with a check that:

- the weight of d is 1 (so the random assignment can't fail);
- for all elements u in the support of d, the result of substituting u for x{i}—where i is the selected side—in the conclusion's original postcondition holds.

For example, if the current goal is

```
Type variables: <none>

&1 (left ) : M.f [programs are in sync]
&2 (right) : M.f

pre = true

(1)  x <$ {0,1}

post = ={x}
```

then running

```
rnd{1}.
```

produces the (false!) goal

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : M.f

pre = true

                        (1)  x <$ {0,1}

post = is_lossless {0,1} && forall (x0 : bool), x0 \in {0,1} => x0 = x{2}
```

Syntax: `rnd`. If the conclusion is an HL statement judgement whose program ends with a random assignment, then consume that random assignment, replacing the conclusion's postcondition by the possibilistic weakest precondition of the random assignment.

For example, if the current goal is

```
Type variables: <none>

Context : M.f

pre = true

(1)  y <- 2
(2)  x <$ [1..10]

post = 3 <= x + y <= 12
```

then running

```
rnd.
```

produces the goal

```
Type variables: <none>

Context : M.f

pre = true
```

```
(1)  y <- 2

post = forall (x0 : int), x0 \in [1..10] => 3 <= x0 + y <= 12
```

Syntax: `rnd` | `rnd` E. In pHL, compute the probabilistic weakest precondition of a random sampling with respect to event E. When E is not specified, it is inferred from the current postcondition.

`if`

Syntax: `if`. If the goal's conclusion is a pRHL statement judgement whose programs both *begin* with `if` statements, reduce the goal to three subgoals:

- One whose conclusion is the ambient logic formula asserting that the equivalence of the boolean expressions of the `if` statements in their respective memories holds given that the statement judgement's precondition holds in those memories.
- One in which the `if` statements have been replaced by their "then" parts, and where the assertion of the truth of the first `if` statement's boolean expression in the first program's memory has been added to the conclusion's precondition.
- One in which the `if` statements have been replaced by their "else" parts, and where the assertion of the falsity of the first `if` statement's boolean expression in the first program's memory has been added to the conclusion's precondition.

For example, if the current goal is

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f

pre = ={x, y}

if (y < x) {                 (1--)  if (y < x) {
  z <- x - y                 (1.1)    z <- x - y + 1
} else {                     (1--)  } else {
  z <- y - x                 (1?1)    z <- y - x + 1
}                            (1--)  }
z <- z * 2                   (2--)

post = 0 <= z{1} /\ z{1} = 2 * (z{2} - 1)
```

then running

```
if.
```

produces the goals

```
Type variables: <none>

forall &1 &2, ={x, y} => y{1} < x{1} <=> y{2} < x{2}
```

and

```
Type variables: <none>

&1 (left ) : M.f
```

```
&2 (right) : N.f

pre = ={x, y} /\ y{1} < x{1}

z <- x - y                (1)  z <- x - y + 1
z <- z * 2                (2)

post = 0 <= z{1} /\ z{1} = 2 * (z{2} - 1)
```

and

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f

pre = ={x, y} /\ ! y{1} < x{1}

z <- y - x                (1)  z <- y - x + 1
z <- z * 2                (2)

post = 0 <= z{1} /\ z{1} = 2 * (z{2} - 1)
```

Syntax: `if{1} | if{2}`. If the goal's conclusion is a pRHL judgement in which the first statement of the specified program is an `if` statement, then reduce the goal to two subgoals:

- One where the `if` statement has been replaced by its "then" part, and the precondition has been augmented by the assertion that the `if` statement's boolean expression is true in the specified program's memory.
- One where the `if` statement has been replaced by its "else" part, and the precondition has been augmented by the assertion that the `if` statement's boolean expression is false in the specified program's memory.

For example, if the current goal is

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f

pre = ={x, y}

if (y < x) {              (1--)  if (y < x) {
  z <- x - y              (1.1)    z <- x - y + 1
} else {                  (1--)  } else {
  z <- y - x              (1?1)    z <- y - x + 1
}                         (1--)  }
z <- z * 2                (2--)

post = 0 <= z{1} /\ z{1} = 2 * (z{2} - 1)
```

then running

```
if{1}.
```

produces the goals

```
Type variables: <none>
```

```
&1 (left ) : M.f
&2 (right) : N.f

pre = ={x, y} /\ y{1} < x{1}

z <- x - y                (1--)  if (y < x) {
                          (1.1)    z <- x - y + 1
                          (1--)  } else {
                          (1?1)    z <- y - x + 1
                          (1--)  }
z <- z * 2                (2--)

post = 0 <= z{1} /\ z{1} = 2 * (z{2} - 1)
```

and

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f

pre = ={x, y} /\ ! y{1} < x{1}

z <- y - x                (1--)  if (y < x) {
                          (1.1)    z <- x - y + 1
                          (1--)  } else {
                          (1?1)    z <- y - x + 1
                          (1--)  }
z <- z * 2                (2--)

post = 0 <= z{1} /\ z{1} = 2 * (z{2} - 1)
```

Syntax: `if`. If the goal's conclusion is an HL judgement whose first statement is an `if` statement, then reduce the goal to two subgoals:

- One where the `if` statement has been replaced by its "then" part, and the precondition has been augmented by the assertion that the `if` statement's boolean expression is true.
- One where the `if` statement has been replaced by its "else" part, and the precondition has been augmented by the assertion that the `if` statement's boolean expression is false.

For example, if the current goal is

```
Type variables: <none>

Context : M.f

pre = true

(1--)  if (y < x) {
(1.1)    z <- x - y
(1--)  } else {
(1?1)    z <- y - x
(1--)  }
(2--)  z <- z * 2

post = 0 <= z
```

then running

```
if.
```

produces the goals

```
Type variables: <none>

--------------------------------------------------------------------
Context : M.f

pre = y < x

(1)  z <- x - y
(2)  z <- z * 2

post = 0 <= z
```

and

```
Type variables: <none>

--------------------------------------------------------------------
Context : M.f

pre = ! y < x

(1)  z <- y - x
(2)  z <- z * 2

post = 0 <= z
```

`while`

Syntax: `while` I. Here I is an *invariant* (formula), which may reference variables of the two programs, interpreted in their memories. If the goal's conclusion is a pRHL statement judgement whose programs both end with `while` statements, reduce the goal to two subgoals whose conclusions are pRHL statement judgements:

- One whose first and second programs are the bodies of the first and second while statements, whose precondition is the conjunction of I and the while statements' boolean expressions (the first of which is interpreted in memory `&1`, and the second of which is interpreted in `&2`) and whose postcondition is the conjunction of I and the assertion that the while statements' boolean expressions (interpreted in the appropriate memories) are equivalent.

- One whose precondition is the original goal's precondition, whose first and second programs are all the results of removing the while statements from the two programs, and whose postcondition is the conjunction of:

  - the conjunction of I and the assertion that the while statements' boolean expressions are equivalent; and

  - the assertion that, for all values of the variables *modified* by the while statements, if the while statements' boolean expressions don't hold, but I holds, then the original goal's postcondition holds (in I, the while statements' boolean expressions, and the postcondition, variables modified by the while statements are replaced by universally quantified identifiers; otherwise, the boolean expressions are interpreted in the program's respective memories, and the memory references of I and the postcondition are maintained).

For example, if the current goal is

```
Type variables: <none>

n : int
------------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : N.f

pre = ={x, y} /\ y{1} = n

z <- 0                     (1--)  z <- 1
while (0 < x) {            (2--)  i <- 1
  z <- z + 1               (2.1)
  x <- x - 1               (2.2)
}                          (2--)
                           (3--)  while (i <= x) {
                           (3.1)    z <- z + 2
                           (3.2)    i <- i + 1
                           (3--)  }

post = (z{1} + y{1} - n) * 2 + 1 = z{2} + y{2} - n
```

then running

```
while (x{1} - 1 = x{2} - i{2} /\ z{1} * 2 + 1 = z{2}).
```

produces the goals

```
Type variables: <none>

n : int
------------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : N.f

pre =
  (x{1} - 1 = x{2} - i{2} /\ z{1} * 2 + 1 = z{2}) /\
  0 < x{1} /\ i{2} <= x{2}

z <- z + 1                 (1)  z <- z + 2
x <- x - 1                 (2)  i <- i + 1

post =
  (x{1} - 1 = x{2} - i{2} /\ z{1} * 2 + 1 = z{2}) /\
  (0 < x{1} <=> i{2} <= x{2})
```

and

```
Type variables: <none>

n : int
------------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : N.f

pre = ={x, y} /\ y{1} = n

z <- 0                     (1)  z <- 1
                           (2)  i <- 1

post =
  ((x{1} - 1 = x{2} - i{2} /\ z{1} * 2 + 1 = z{2}) /\
```

```
   (0 < x{1} <=> i{2} <= x{2})) /\
  forall (x_L z_L i_R z_R : int),
    ! 0 < x_L =>
    ! i_R <= x{2} =>
    x_L - 1 = x{2} - i_R /\ z_L * 2 + 1 = z_R =>
    (z_L + y{1} - n) * 2 + 1 = z_R + y{2} - n
```

Syntax: `while{1}` I v `|` `while{2}` I v. Here I is an *invariant* (formula) and v is a *termination variant* integer expression, both of which may reference variables of the two programs, interpreted in their memories. If the goal's conclusion is a pRHL statement judgement whose designated program (1 or 2) ends with a `while` statement, reduce the goal to two subgoals;

- One whose conclusion is a pHL statement judgement, saying that running the body of the while statement in a memory in which I holds and the while statement's boolean expression is true is guaranteed to result in termination in a memory in which I holds and in which the value of the variant expression v has decreased by at least 1. (More precisely, the pHL statement judgment is universally quantified by the memory of the non-designated program and the initial value of v. References to the variables of the nondesignated program in I and v are interpreted in this memory; reference to the variables of the designed program have their memory references removed.)
- One whose conclusion is a pRHL statement judgement whose precondition is the original goal's precondition, whose designated program is the result of removing the while statement from the original designated program, whose other program is unchanged, and whose postcondition is the conjunction of I and the assertion that, for all values of the variables modified by the while statement, that the conjunction of the following formulas holds:
  - the assertion that, if I holds, but the variant expression v is not positive, then the while statement's boolean expression is false;
  - the assertion that, if the while statement's boolean expression doesn't hold, but I holds, then the original goal's postcondition holds.

For example, if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : N.f

pre = ={n} /\ 0 <= n{1}

x <- 0                      (1--)
i <- 0                      (2--)
while (i < n) {             (3--)
  x <- x + (i + 1)          (3.1)
  i <- i + 1                (3.2)
}                           (3--)

post = x{1} <= n{2} * n{2}
```

then running

```
while{1} (0 <= i{1} <= n{1} /\ x{1} <= i{1} * i{1}) (n{1} - i{1}).
```

produces the goals

```
Type variables: <none>
```

```
forall _ (z : int),
  phoare[ x <- x + (i + 1); i <- i + 1 :
          ((0 <= i <= n /\ x <= i * i) /\ i < n) /\ n - i = z ==>
          (0 <= i <= n /\ x <= i * i) /\ n - i < z] = 1%r
```

and

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f

pre = ={n} /\ 0 <= n{1}

x <- 0                    (1)
i <- 0                    (2)

post =
  (0 <= i{1} <= n{1} /\ x{1} <= i{1} * i{1}) /\
  forall (i_L x_L : int),
    (0 <= i_L <= n{1} /\ x_L <= i_L * i_L =>
     n{1} - i_L <= 0 => ! i_L < n{1}) /\
    (! i_L < n{1} =>
     0 <= i_L <= n{1} /\ x_L <= i_L * i_L => x_L <= n{2} * n{2})
```

Syntax: `while` I. Here I is an *invariant* (formula), which may reference variables of the program. If the goal's conclusion is an HL statement judgement ending with a `while` statement, reduce the goal to two subgoals whose conclusions are HL statement judgements:

- One whose program is the body of the while statement, whose precondition is the conjunction of I and the while statement's boolean expression, and whose postcondition is I.
- One whose precondition is the original goal's precondition, whose program is the result of removing the while statement from the original program, and whose postcondition is the conjunction of:
  - I; and
  - the assertion that, for all values of the variables *modified* by the while statement, if the while statement's boolean expression doesn't hold, but I holds, then the original goal's postcondition holds (in I, the while statement's boolean expression, and the postcondition, variables modified by the while statement are replaced by universally quantified identifiers).

For example, if the current goal is

```
Type variables: <none>

m : int

Context : M.f

pre = m = n /\ 0 <= n

(1--)  x <- 0
(2--)  i <- 0
(3--)  while (i < n) {
(3.1)    x <- x + (i + 1)
```

```
(3.2)    i <- i + 1
(3--)  }

post = x <= m * m
```

then running

```
while (0 <= i <= n /\ x <= i * i).
```

produces the goals

```
Type variables: <none>

m : int
------------------------------------------------------------------
Context : M.f

pre = (0 <= i <= n /\ x <= i * i) /\ i < n

(1)  x <- x + (i + 1)
(2)  i <- i + 1

post = 0 <= i <= n /\ x <= i * i
```

and

```
Type variables: <none>

m : int
------------------------------------------------------------------
Context : M.f

pre = m = n /\ 0 <= n

(1)  x <- 0
(2)  i <- 0

post =
  (0 <= i <= n /\ x <= i * i) /\
  forall (i0 x0 : int),
    ! i0 < n => 0 <= i0 <= n /\ x0 <= i0 * i0 => x0 <= m * m
```

Syntax: `while` I v. pHL version...

`call`

When describing the variants of this tactic, we ony consider procedure call assignments whose left-hand sides consist of single identifiers. The generalization to multiple assignment, when values of tuple types are returned, is straightforward.

Syntax: `call (_ :` P `==>` Q`)`. If the goal's conclusion is a pRHL statement judgement whose programs end with procedure calls or procedure call assignments (resp., an HL statement judgement whose program ends with a procedure call or procedure call assignment), then generate two subgoals:

- One whose conclusion is a pRHL judgement (resp., HL judgement) whose precondition is P, whose procedures are the procedures being called (resp., procedure is the procedure being called), and whose postcondition is Q.
- One whose conclusion is a pRHL statement judgement (resp., HL statement judgement) whose precondition is the original goal's precondition, whose programs are (resp., program is) the result of removing the procedure calls (resp., call) from the programs (resp., program), and whose postcondition is the conjunction of

– the result of replacing the procedures' (resp., procedure's) parameter(s) by their actual argument(s) in P; and
– the assertion that, for all values of the global variable(s) modified by the procedures (resp., procedure) and the results (resp., result) of the procedure calls (resp., procedure call), if Q holds (where these quantified identifiers have been substituted for the modified variables and procedure results), then the original goal's postcondition holds (where the modified global variables and occurrences of the variables (resp., variable) (if any) to which the results of the procedure calls are (resp., result of the procedure call is) assigned have been replaced by the appropriate quantified identifiers).

For example, if the current goal is

```
Type variables: <none>
--------------------------------------------------------------------
&1 (left ) : M.g
&2 (right) : N.g

pre = ={u}

M.x <- 5                    (1)  N.x <- -5
z <@ M.f(u)                 (2)  z <@ N.f(u)

post = z{1} + M.x{1} = z{2} - N.x{2}
```

and the procedures `M.f` and `N.f` have a single parameter, `y`, then running

```
call (_ : ={y} /\ M.x{1} = -N.x{2} ==> ={res} /\ M.x{1} = -N.x{2}).
```

produces the goals

```
Type variables: <none>
--------------------------------------------------------------------
pre = ={y} /\ M.x{1} = -N.x{2}

    M.f ~ N.f

post = ={res} /\ M.x{1} = -N.x{2}
```

and

```
Type variables: <none>
--------------------------------------------------------------------
&1 (left ) : M.g
&2 (right) : N.g

pre = ={u}

M.x <- 5                    (1)  N.x <- -5

post =
  (={u} /\ M.x{1} = -N.x{2}) &&
  forall (result_L result_R x_L x_R : int),
    result_L = result_R /\ x_L = -x_R =>
    result_L + x_L = result_R - x_R
```

Alternatively, a proof term whose conclusion is a pRHL or HL judgement involving the procedure(s) called at the end(s) of the program(s) may be supplied as the argument to `call`, in which case only the second subgoal need be generated.

For example, in the start-goal of the preceding example, if the lemma `M_N_f` is

```
lemma M_N_f :
  equiv[M.f ~ N.f :
        ={y} /\ M.x{1} = -N.x{2} ==> ={res} /\ M.x{1} = -N.x{2}].
```

then running

```
call M_N_f.
```

produces the goal

```
Type variables: <none>

--------------------------------------------------------------------
&1 (left ) : M.g
&2 (right) : N.g

pre = ={u}

M.x <- 5                    (1)  N.x <- -5

post =
  (={u} /\ M.x{1} = -N.x{2}) &&
  forall (result_L result_R x_L x_R : int),
    result_L = result_R /\ x_L = -x_R =>
    result_L + x_L = result_R - x_R
```

Syntax: `call{1} (_ :` P `==>` Q`) | call{2} (_ :` P `==>` Q`)`. If the goal's conclusion is a pRHL statement judgement whose designated program ends with a procedure call, then generate two subgoals:

- One whose conclusion is a pHL judgement whose precondition is P, whose procedure is the procedure being called, whose postcondition is Q, and whose bound part specifies equality with probability 1. (Consequently, P and Q may not mention `&1` and `&2`.)
- One whose conclusion is a pRHL statement judgement whose precondition is the original goal's precondition, whose programs are the result of removing the procedure call from the designated program, and leaving the other program unchanged, and whose postcondition is the conjunction of
  - the result of replacing the procedure's parameter(s) by their actual argument(s) in P; and
  - the assertion that, for all values of the global variable(s) modified by the procedure and the result of the procedure call, if Q holds (where these quantified identifiers have been substituted for the modified variables and procedure result), then the original goal's postcondition holds (where the modified global variables and occurrences the variable (if any) to which the result of the procedure call is assigned have been replaced by the appropriate quantified identifiers).

For example, if the current goal is

```
Type variables: <none>

x2: int

--------------------------------------------------------------------
&1 (left ) : M.g
&2 (right) : N.g

pre = x2 = N.x{2} /\ N.x{2} = u{2} /\ u{1} %% 2 = u{2} %% 2
```

```
M.x <- u                    (1)
M.f(7)                      (2)

post = (M.x{1} %% 2 = 0) = (N.x{2} %% 2 = 0)
```

then running

```
call{1} (_ : M.x %% 2 = x2 %% 2 ==> M.x %% 2 = x2 %% 2).
```

produces the goals

```
Type variables: <none>

x2: int
-----------------------------------------------------------------------
pre = M.x %% 2 = x2 %% 2

    M.f
    [=] 1%r

post = M.x %% 2 = x2 %% 2
```

and

```
Type variables: <none>

x2: int
-----------------------------------------------------------------------
&1 (left ) : M.g
&2 (right) : N.g

pre = x2 = N.x{2} /\ N.x{2} = u{2} /\ u{1} %% 2 = u{2} %% 2

M.x <- u                    (1)

post =
  M.x{1} %% 2 = x2 %% 2 &&
  forall (x_L : int),
    x_L %% 2 = x2 %% 2 => (x_L %% 2 = 0) = (N.x{2} %% 2 = 0)
```

Alternatively, a proof term whose conclusion is a pHL judgement specifying equality with probability 1 and involving the procedure called at the end of the designated program may be supplied as the argument to `call`, in which case only the second subgoal need be generated.

For example, in the start-goal of the preceding example, if the lemma `M_f` is

```
lemma M_f (z : int) :
  phoare [M.f : M.x %% 2 = z %% 2 ==> M.x %% 2 = z %% 2] = 1%r.
```

then running

```
call{1} (M_f x2).
```

produces the goal

```
Type variables: <none>

x2: int
-----------------------------------------------------------------------
&1 (left ) : M.g
&2 (right) : N.g
```

```
pre = x2 = N.x{2} /\ N.x{2} = u{2} /\ u{1} %% 2 = u{2} %% 2

M.x <- u                   (1)

post =
  M.x{1} %% 2 = x2 %% 2 &&
  forall (x_L : int),
    x_L %% 2 = x2 %% 2 => (x_L %% 2 = 0) = (N.x{2} %% 2 = 0)
```

Syntax: `call (_ : I)`. If the conclusion is a pRHL statement judgement whose programs end with calls to *concrete* procedures (resp., an HL statement judgement whose program ends with a call to a concrete procedure), then use the specification argument to `call` generated from the *invariant* I, and automatically apply `proc` to its first subgoal. In the pRHL case, its precondition will assume equality of the procedures' parameters, and its postcondition will assert equality of the results of the procedure calls.

For example, if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.g
&2 (right) : N.g

pre = ={u}

M.x <- 5                   (1)  N.x <- -5
z <@ M.f(u)                (2)  z <@ N.f(u)

post = z{1} + M.x{1} = z{2} - N.x{2}
```

and modules `M` and `N` contain

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

and

```
  var x : int
  proc f(y : int) : int = {
    x <- x - y;
    return -x;
  }
```

respectively, then running

```
call (_ : M.x{1} = -N.x{2}).
```

produces the goals

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : N.f

pre = ={y} /\ M.x{1} = -N.x{2}

M.x <- M.x + y             (1)  N.x <- N.x - y
```

```
post = M.x{1} = -N.x{2} /\ M.x{1} = -N.x{2}
```

and

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.g
&2 (right) : N.g

pre = ={u}

M.x <- 5                  (1)  N.x <- -5

post =
  (={u} /\ M.x{1} = -N.x{2}) &&
  forall (result_L result_R x_L x_R : int),
    result_L = result_R /\ x_L = -x_R =>
    result_L + x_L = result_R - x_R
```

Syntax: `call (_ : I)`. If the conclusion is a pRHL statement judgement whose programs end with calls of the same *abstract* procedure (resp., an HL statement judgement whose program ends with a call to an abstract procedure), then use the specification argument to `call` generated from the *invariant* I, and automatically apply `proc` I to its first subgoal, pruning the first two subgoals the application generates, because their conclusions consist of ambient logic formulas that are true by construction. In the pRHL case, its precondition will assume equality of the procedure's parameters and of the global variables of the module containing the procedure, and its postcondition will assume equality of the results of the procedure calls and of the global variables of the containing module.

For example, given the declarations

```
module type OR = {
  proc init(i : int) : unit
  proc f1() : unit
  proc f2() : unit
}.

module Or : OR = {
  var x : int
  proc init(i : int) : unit = {
    x <- i;
  }
  proc f1() : unit = {
    x <- x + 2;
  }
  proc f2() : unit = {
    x <- x - 2;
  }
}.

module type T(O : OR) = {
  proc g() : unit {O.f1 O.f2}
}.
```

if the current goal is

```
Type variables: <none>

Adv: T{Or}
```

```
&1 (left ) : M(Adv).h
&2 (right) : N(Adv).h

pre = ={y, glob Adv} /\ Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0

Adv(Or).g()                (1)  Adv(Or).g()

post = Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0
```

then running

```
call (_ : Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0).
```

produces the goals

```
Type variables: <none>

Adv: T{Or}
------------------------------------------------------------------------
pre = true /\ Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0

    Or.f1 ~ Or.f1

post = ={res} /\ Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0
```

and

```
Type variables: <none>

Adv: T{Or}
------------------------------------------------------------------------
pre = true /\ Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0

    Or.f2 ~ Or.f2

post = ={res} /\ Or.x{1} %% 2 = 0 /\ Or.x{2} %% 2 = 0
```

Syntax: `call (_ :` B, I`)`. If the conclusion is a pRHL statement judgement whose programs end with calls of the same *abstract* procedure, then use the specification argument to `call` generated from the *bad event* B and *invariant* I, and automatically apply `proc` B I to its first subgoal, pruning the first two subgoals the application generates, because their conclusions consist of ambient logic formulas that are true by construction, and pruning the next goal (showing the losslessness of the abstract procedure given the losslessness of the abstract oracles it uses), if trivial suffices to solve it. The specification's precondition will assume equality of the procedure's parameters and of the global variables of the module containing the procedure as well as I, and its postcondition will assert I and the equality of the results of the procedure calls and of the global variables of the containing module—but only when B does not hold.

For example, given the declarations

```
module type OR = {
  proc init() : unit
  proc qry(x : int) : int
}.

op low : int = -100.
op upp : int = 100.

module Or1 : OR = {
  var qry, rsp : int
```

```
  var queried : bool

  proc init() = {
    qry <$ [low .. upp]; rsp <$ [low .. upp];
    queried <- false;
  }

  proc qry(x : int) : int = {
    var y : int;
    if (x = qry) {
      y <- rsp;
      queried <- true;
    } else {
      y <$ [low .. upp];
    }
    return y;
  }
}.

module Or2 : OR = {
  var qry : int
  var queried : bool

  proc init() = {
    qry <$ [low .. upp];
    queried <- false;
  }

  proc qry(x : int) : int = {
    var y : int;
    y <$ [low .. upp];
    queried <- queried \/ x = qry;
    return y;
  }
}.

module type ADV(O : OR) = {
  proc * f() : bool {O.qry}
}.
```

if the current goal is

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------------
&1 (left ) : M(Adv).h
&2 (right) : N(Adv).h

pre = Or1.qry{1} = Or2.qry{2}

b <@ Adv(Or1).f()          (1)  b <@ Adv(Or2).f()

post = !Or2.queried{2} => ={b}
```

then running

```
call (_ : Or2.queried, (Or1.qry{1} = Or2.qry{2})).
```

produces the goals

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------
pre = !Or2.queried{2} /\ ={x} /\ Or1.qry{1} = Or2.qry{2}

    Or1.qry ~ Or2.qry

post = !Or2.queried{2} => ={res} /\ Or1.qry{1} = Or2.qry{2}
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------
forall &2, Or2.queried{2} => islossless Or1.qry
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------
forall _,
  phoare[ Or2.qry : Or2.queried /\ true ==> Or2.queried /\ true] = 1%r
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------
&1 (left ) : M(Adv).h
&2 (right) : N(Adv).h

pre = Or1.qry{1} = Or2.qry{2}


post =
  (!Or2.queried{2} => true /\ Or1.qry{1} = Or2.qry{2}) &&
  forall (result_L result_R : bool) (Adv_L Adv_R : (glob Adv))
    (queried_R : bool),
    (!queried_R =>
     result_L = result_R /\
     Adv_L = Adv_R /\ Or1.qry{1} = Or2.qry{2}) =>
    !queried_R => result_L = result_R
```

Syntax: `call (_ : B, I, J)`. If the conclusion is a pRHL statement judgement whose programs end with calls of the same *abstract* procedure, then use the specification argument to `call` generated from the *bad event* B and *invariants* I and J, and automatically apply `proc` B I J to its first subgoal, pruning the first two subgoals the application generates, because their conclusions consist of ambient logic formulas that are true by construction, and pruning the next goal (showing the losslessness of the abstract procedure given the losslessness of the abstract oracles it uses), if trivial suffices to solve it. The specification's precondition will assume equality of the procedure's parameters and of the global variables of the module containing the procedure as well as I, and its postcondition will assert

- I and the equality of the results of the procedure calls and of the global variables of the containing module—if B does not hold; and
- J—if B does hold.

For example, given the declarations of the preceding example if the current goal is

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------
&1 (left ) : M(Adv).h
&2 (right) : N(Adv).h

pre = Or1.qry{1} = Or2.qry{2} /\ Or1.queried{1} = Or2.queried{2}

b <@ Adv(Or1).f()          (1)   b <@ Adv(Or2).f()

post =
  Or1.queried{1} = Or2.queried{2} /\ (!Or2.queried{2} => ={b})
```

then running

```
call (_ :
        Or2.queried,
        (Or1.qry{1} = Or2.qry{2} /\ Or1.queried{1} = Or2.queried{2}),
        (Or1.queried{1} = Or2.queried{2})).
```

produces the goals

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
------------------------------------------------------------------
pre =
  !Or2.queried{2} /\
  ={x} /\
  Or1.qry{1} = Or2.qry{2} /\ Or1.queried{1} = Or2.queried{2}

    Or1.qry ~ Or2.qry

post =
  if Or2.queried{2} then Or1.queried{1} = Or2.queried{2}
  else
    ={res} /\
    Or1.qry{1} = Or2.qry{2} /\ Or1.queried{1} = Or2.queried{2}
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
```

```
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
--------------------------------------------------------------------
forall &2,
  Or2.queried{2} =>
  phoare[ Or1.qry :
           Or1.queried = Or2.queried{2} ==>
           Or1.queried = Or2.queried{2}] = 1%r
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
--------------------------------------------------------------------
forall &1,
  phoare[ Or2.qry :
           Or2.queried /\ Or1.queried{1} = Or2.queried ==>
           Or2.queried /\ Or1.queried{1} = Or2.queried] = 1%r
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
            islossless O.qry => islossless Adv(O).f
--------------------------------------------------------------------
&1 (left ) : M(Adv).h
&2 (right) : N(Adv).h

pre = Or1.qry{1} = Or2.qry{2} /\ Or1.queried{1} = Or2.queried{2}


post =
  (if Or2.queried{2} then Or1.queried{1} = Or2.queried{2}
   else
     true /\
     Or1.qry{1} = Or2.qry{2} /\ Or1.queried{1} = Or2.queried{2}) &&
  forall (result_L result_R : bool) (Adv_L : (glob Adv))
    (queried_L : bool) (Adv_R : (glob Adv)) (queried_R : bool),
    if queried_R then queried_L = queried_R
    else
      result_L = result_R /\
      Adv_L = Adv_R /\
      Or1.qry{1} = Or2.qry{2} /\ queried_L = queried_R =>
    queried_L = queried_R /\ (!queried_R => result_L = result_R)
```

3.4.2 Tactics for Transforming Programs

Unless otherwise specified, the tactics of this subsection only apply to goals whose conclusions are pRHL, pHL or HL statement judgements, reducing such a goal to a single subgoal in which only the program(s) of those statement judgements have changed.

Many of these tactics take *code positions* consisting of a sequence of positive numerals separated by dots. E.g., `2.1.3` says to go to the statement 2 of the program, then to substatement 1 of it, then to sub-substatement 3 of it. We use the variable c to range over code positions.

⊚ **swap**

All versions of the tactic work for pRHL (an optional side can be given), pHL and HL statement judgements. We'll describe their operation in terms of a single program (list of statements).

Syntax: `swap` $n\ m\ l$. Fails unless $1 \leq n < m \leq l$ and the program has at least l statements. Swaps the statement block from positions n through $m-1$ with the statement block from m through l, failing if these blocks of statements aren't independent.

Syntax: `swap` $[n\,..\,m]\ k$. Fails unless $1 \leq n \leq m$ and the program has at least m statements.

- If k is non-negative, move the statement block from n through m forward k positions, failing if the program doesn't have at least $m+k$ statements or if the swapped statements blocks aren't independent.
- If k is negative, move the statement block from n through m backward $-k$ positions, failing if $n+k<1$ or if the swapped statement blocks aren't independent.

Syntax: `swap` $n\ \ k$. Equivalent to `swap` $[n\,..\,n]\ k$.

Syntax: `swap` k. If k is non-negative, equivalent to `swap` $1\ k$. If k is negative, equivalent to `swap` $n\ k$, where n is the length of the program.

For example, suppose the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.f [programs are in sync]
&2 (right) : M.f

pre = ={M.x, M.y, M.z, M.w}

(1)  M.x <- true
(2)  M.y <- false
(3)  M.z <- true
(4)  M.w <- false

post = ={M.x, M.y, M.z, M.w}
```

Then running

```
swap 1 3 4.
```

produces goal

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.f [programs are in sync]
&2 (right) : M.f

pre = ={M.x, M.y, M.z, M.w}

(1)  M.z <- true
(2)  M.w <- false
(3)  M.x <- true
(4)  M.y <- false

post = ={M.x, M.y, M.z, M.w}
```

From which running

```
swap 2 2.
```

produces goal

```
Type variables: <none>

________________________________________________________________________________
&1 (left ) : M.f [programs are in sync]
&2 (right) : M.f

pre = ={M.x, M.y, M.z, M.w}

(1)  M.z <- true
(2)  M.x <- true
(3)  M.y <- false
(4)  M.w <- false

post = ={M.x, M.y, M.z, M.w}
```

From which running

```
swap{1} [3 .. 4] -1.
```

produces goal

```
Type variables: <none>

________________________________________________________________________________
&1 (left ) : M.f
&2 (right) : M.f

pre = ={M.x, M.y, M.z, M.w}

M.z <- true                (1)  M.z <- true
M.y <- false               (2)  M.x <- true
M.w <- false               (3)  M.y <- false
M.x <- true                (4)  M.w <- false

post = ={M.x, M.y, M.z, M.w}
```

From which running

```
swap 2.
```

produces goal

```
Type variables: <none>

________________________________________________________________________________
&1 (left ) : M.f
&2 (right) : M.f

pre = ={M.x, M.y, M.z, M.w}

M.y <- false               (1)  M.x <- true
M.w <- false               (2)  M.y <- false
M.z <- true                (3)  M.z <- true
M.x <- true                (4)  M.w <- false

post = ={M.x, M.y, M.z, M.w}
```

From which running

```
swap{2} -1.
```

produces goal

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : M.f

pre = ={M.x, M.y, M.z, M.w}

M.y <- false              (1)  M.x <- true
M.w <- false              (2)  M.y <- false
M.z <- true               (3)  M.w <- false
M.x <- true               (4)  M.z <- true

post = ={M.x, M.y, M.z, M.w}
```

`inline`

Syntax: `inline` $M_1 . p_1 \cdots M_n . p_n$. Inline the selected *concrete* procedures in both programs, with pRHL, and in the program, with HL and pHL, until no more inlining of these procedures is possible.
To inline a procedure call, the procedure's parameters are assigned the values of their arguments (fresh parameter identifiers are used, as necessary, to avoid naming conflicts). This is followed by the body of the procedure. Finally, the procedure's return value is assigned to the identifiers (if any) to which the procedure call's result is assigned.

Syntax: `inline{1}` $M_1 . p_1 \cdots M_n . p_n$ `|` `inline{2}` $M_1 . p_1 \cdots M_n . p_n$. Do the inlining in just the first or second program, in the pRHL case.

Syntax: `inline*` `|` `inline{1}*` `|` `inline{2}*`. Inline all concrete procedures, continuing until no more inlining is possible.

Syntax: `inline` *occs* $M . p$ `|` `inline{1}` *occs* $M . p$ `|` `inline{2}` *occs* $M . p$. Inline just the specified occurrences of $M.p$, where *occs* is a parenthesized nonempty sequence of positive numbers $(n_1 \cdots n_l)$. E.g., `(1 3)` means the first and third occurrences of the procedure. In the pRHL case, a side `{1}` or `{2}` must be specified.

For example, given the declarations

```
module M = {
  var y : int
  proc f(x : int) : int = {
    x <- x + 1;
    return x * 2;
  }
  proc g(x : int) : bool = {
    y <@ f(x - 1);
    return x + y + 1 = 3;
  }
  proc h(x : int) : bool = {
    var b : bool;
    b <@ g(x + 1);
    return !b;
  }
}.
```

if the current goal is

```
Type variables: <none>
```

```
&1 (left ) : M.h [programs are in sync]
&2 (right) : M.h

pre = ={x}

(1)  b <@ M.g(x + 1)

post = (!b{1}) = !b{2}
```

then running

```
inline M.g.
```

produces the goal

```
Type variables: <none>

&1 (left ) : M.h [programs are in sync]
&2 (right) : M.h

pre = ={x}

(1)  x0 <- x + 1
(2)  M.y <@ M.f(x0 - 1)
(3)  b <- x0 + M.y + 1 = 3

post = (!b{1}) = !b{2}
```

From which running

```
inline{2} M.f.
```

produces the goal

```
Type variables: <none>

&1 (left ) : M.h
&2 (right) : M.h

pre = ={x}

x0 <- x + 1                (1)  x0 <- x + 1
M.y <@ M.f(x0 - 1)         (2)  x1 <- x0 - 1
b <- x0 + M.y + 1 = 3      (3)  x1 <- x1 + 1
                           (4)  M.y <- x1 * 2
                           (5)  b <- x0 + M.y + 1 = 3

post = (!b{1}) = !b{2}
```

From which running

```
inline M.f.
```

produces the goal

```
Type variables: <none>

&1 (left ) : M.h [programs are in sync]
```

```
&2 (right) : M.h

pre = ={x}

(1)  x0 <- x + 1
(2)  x1 <- x0 - 1
(3)  x1 <- x1 + 1
(4)  M.y <- x1 * 2
(5)  b <- x0 + M.y + 1 = 3

post = (!b{1}) = !b{2}
```

And, if the current goal is

```
Type variables: <none>

&1 (left ) : M.h [programs are in sync]
&2 (right) : M.h

pre = ={x}

(1)  b <@ M.g(x + 1)

post = (!b{1}) = !b{2}
```

then running

```
inline*.
```

produces the goal

```
Type variables: <none>

&1 (left ) : M.h [programs are in sync]
&2 (right) : M.h

pre = ={x}

(1)  x0 <- x + 1
(2)  x1 <- x0 - 1
(3)  x1 <- x1 + 1
(4)  M.y <- x1 * 2
(5)  b <- x0 + M.y + 1 = 3

post = (!b{1}) = !b{2}
```

rcondt

Syntax: `rcondt` n. If the goal's conclusion is an HL statement judgement whose nth statement is an `if` statement, reduce the goal to two subgoals.

- One whose concludion is an HL statement judgement whose precondition is the original goal's precondition, program is the first $n-1$ statements of the original goal's program, and postcondition is the boolean expression of the `if` statement.
- One whose conclusion is an HL statement judgement that's the same as that of the original goal except that the `if` statement has been replaced by its `then` part.

For example, if the current goal is

```
Type variables: <none>

Context : M.f

pre = true

(1--)  x <$ {0,1}
(2--)  y <- x ^^ x
(3--)  if (!y) {
(3.1)    z <- true
(3--)  } else {
(3?1)    z <- false
(3--)  }

post = z
```

then running

```
rcondt 3.
```

produces the goals

```
Type variables: <none>

Context : M.f

pre = true

(1)  x <$ {0,1}
(2)  y <- x ^^ x

post = !y
```

and

```
Type variables: <none>

Context : M.f

pre = true

(1)  x <$ {0,1}
(2)  y <- x ^^ x
(3)  z <- true

post = z
```

Syntax: `rcondt{1}` n | `rcondt{2}` n. If the goal's conclusion is a pRHL statement judgement where the nth statement of the designated program is an `if` statement, reduce the goal to two subgoals.

- One whose conclusion is an HL statement judgement whose precondition is the original goal's precondition, program is the first $n-1$ statements of the original goal's designated program, and postcondition is the boolean expression of the `if` statement. Actually, the HL statement judgement is universally quantified by a memory of the non-designated program, and references in the precondition to variables of the non-designated program are interpreted in that memory.

- One whose conclusion is a pRHL statement judgement that's the same as that of the original goal except that the `if` statement has been replaced by its `then` part.

For example, if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : N.f
&2 (right) : M.f

pre = N.x{1}

                        (1--)  x <$ {0,1}
                        (2--)  y <- x ^^ x
                        (3--)  if (!y) {
                        (3.1)    z <- true
                        (3--)  } else {
                        (3?1)    z <- false
                        (3--)  }

post = N.x{1} = z{2}
```

then running

```
rcondt{2} 3.
```

produces the goals

```
Type variables: <none>

------------------------------------------------------------------------
forall &m, hoare[ x <$ ${0,1}; y <- x ^^ x : N.x{m} ==> !y]
```

and

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : N.f
&2 (right) : M.f

pre = N.x{1}

                        (1)  x <$ {0,1}
                        (2)  y <- x ^^ x
                        (3)  z <- true

post = N.x{1} = z{2}
```

Syntax: `rcondt` n. If the goal's conclusion is an HL statement judgement whose nth statement is a `while` statement, reduce the goal to two subgoals.

- One whose concludion is an HL statement judgement whose precondition is the original goal's precondition, program is the first $n - 1$ statements of the original goal's program, and postcondition is the boolean expression of the `while` statement.

- One whose conclusion is an HL statement judgement that's the same as that of the original goal except that the `while` statement has been replaced by its body.

For example, if the current goal is

```
Type variables: <none>

Context : M.f

pre = true

(1--)  i <- 0
(2--)  while (i < 10) {
(2.1)    i <- i + 1
(2--)  }
(3--)  while (i < 20) {
(3.1)    i <- i + 2
(3--)  }

post = i = 20
```

then running

```
rcondt 3.
```

produces the goals

```
Type variables: <none>

Context : M.f

pre = true

(1--)  i <- 0
(2--)  while (i < 10) {
(2.1)    i <- i + 1
(2--)  }

post = i < 20
```

and

```
Type variables: <none>

Context : M.f

pre = true

(1--)  i <- 0
(2--)  while (i < 10) {
(2.1)    i <- i + 1
(2--)  }
(3--)  i <- i + 2
(4--)  while (i < 20) {
(4.1)    i <- i + 2
(4--)  }

post = i = 20
```

Syntax: `rcondt{1}` n `|` `rcondt{2}` n. If the goal's conclusion is a pRHL statement judgement where the nth statement of the designated program is a `while` statement, reduce the goal to two subgoals.

- One whose conclusion is an HL statement judgement whose precondition is the original goal's precondition, program is the first $n - 1$ statements of the original goal's designated program, and postcondition is the boolean expression of the `while` statement. Actually, the HL statement judgement is universally quantified by a memory of the non-designated program, and references in the precondition to variables of the non-designated program are interpreted in that memory.
- One whose conclusion is a pRHL statement judgement that's the same as that of the original goal except that the `while` statement has been replaced by its body.

For example, if the current goal is

```
Type variables: <none>

--------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : N.f

pre = true

i <- 0                          (1--)  i <- 0
while (i < 10) {                (2--)  while (i < 10) {
  i <- i + 1                    (2.1)    i <- i + 1
}                               (2--)  }
while (i < 20) {                (3--)  i <- i + 2
  i <- i + 2                    (3.1)
}                               (3--)
                                (4--)  while (i < 20) {
                                (4.1)    i <- i + 2
                                (4--)  }

post = ={i}
```

then running

```
rcondt{1} 3.
```

produces the goals

```
Type variables: <none>

--------------------------------------------------------------
forall _, hoare[ i <- 0; while (i < 10) {···} : true ==> i < 20]
```

and

```
Type variables: <none>

--------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : N.f

pre = true

i <- 0                          (1--)  i <- 0
while (i < 10) {                (2--)  while (i < 10) {
  i <- i + 1                    (2.1)    i <- i + 1
}                               (2--)  }
i <- i + 2                      (3--)  i <- i + 2
while (i < 20) {                (4--)  while (i < 20) {
  i <- i + 2                    (4.1)    i <- i + 2
```

```
}                           (4--)  }

post = ={i}
```

rcondf

Syntax: `rcondf` n. If the goal's conclusion is an HL statement judgement whose nth statement is an `if` statement, reduce the goal to two subgoals.

- One whose concludion is an HL statement judgement whose precondition is the original goal's precondition, program is the first $n - 1$ statements of the original goal's program, and postcondition is the negation of the boolean expression of the `if` statement.
- One whose conclusion is an HL statement judgement that's the same as that of the original goal except that the `if` statement has been replaced by its `else` part.

For example, if the current goal is

```
Type variables: <none>

Context : M.f

pre = true

(1--)  x <$ {0,1}
(2--)  y <- x ^^ x
(3--)  if (y) {
(3.1)    z <- true
(3--)  } else {
(3?1)    z <- false
(3--)  }

post = !z
```

then running

```
rcondf 3.
```

produces the goals

```
Type variables: <none>

Context : M.f

pre = true

(1)  x <$ {0,1}
(2)  y <- x ^^ x

post = !y
```

and

```
Type variables: <none>

Context : M.f

pre = true
```

```
(1)  x <$ {0,1}
(2)  y <- x ^^ x
(3)  z <- false

post = !z
```

Syntax: `rcondf{1}` n | `rcondf{2}` n. If the goal's conclusion is a pRHL statement judgement where the nth statement of the designated program is an `if` statement, reduce the goal to two subgoals.

- One whose conclusion is an HL statement judgement whose precondition is the original goal's precondition, program is the first $n-1$ statements of the original goal's designated program, and postcondition is the negation of the boolean expression of the `if` statement. Actually, the HL statement judgement is universally quantified by a memory of the non-designated program, and references in the precondition to variables of the non-designated program are interpreted in that memory.
- One whose conclusion is a pRHL statement judgement that's the same as that of the original goal except that the `if` statement has been replaced by its `else` part.

For example, if the current goal is

```
Type variables: <none>

&1 (left ) : N.f
&2 (right) : M.f

pre = !N.x{1}

                          (1--)  x <$ {0,1}
                          (2--)  y <- x ^^ x
                          (3--)  if (y) {
                          (3.1)    z <- true
                          (3--)  } else {
                          (3?1)    z <- false
                          (3--)  }

post = N.x{1} = z{2}
```

then running

```
rcondf{2} 3.
```

produces the goals

```
Type variables: <none>

forall &m, hoare[ x <$ ${0,1}; y <- x ^^ x : !N.x{m} ==> !y]
```

and

```
Type variables: <none>

&1 (left ) : N.f
&2 (right) : M.f

pre = !N.x{1}
```

```
                              (1)  x <$ {0,1}
                              (2)  y <- x ^^ x
                              (3)  z <- false

post = N.x{1} = z{2}
```

Syntax: `rcondf` n. If the goal's conclusion is an HL statement judgement whose nth statement is a `while` statement, reduce the goal to two subgoals.

- One whose concludion is an HL statement judgement whose precondition is the original goal's precondition, program is the first $n-1$ statements of the original goal's program, and postcondition is the negation of the boolean expression of the `while` statement.
- One whose conclusion is an HL statement judgement that's the same as that of the original goal except that the `while` statement has been removed.

For example, if the current goal is

```
Type variables: <none>

Context : M.f

pre = true

(1--)  i <- 0
(2--)  while (i < 10) {
(2.1)    i <- i + 1
(2--)  }
(3--)  while (i < 10) {
(3.1)    i <- i + 2
(3--)  }

post = i = 10
```

then running

```
rcondf 3.
```

produces the goals

```
Type variables: <none>

Context : M.f

pre = true

(1--)  i <- 0
(2--)  while (i < 10) {
(2.1)    i <- i + 1
(2--)  }

post = ! i < 10
```

and

```
Type variables: <none>
```

```
Context : M.f

pre = true

(1--)  i <- 0
(2--)  while (i < 10) {
(2.1)    i <- i + 1
(2--)  }

post = i = 10
```

Syntax: `rcondf{1}` n `|` `rcondf{2}` n. If the goal's conclusion is a pRHL statement judgement where the nth statement of the designated program is a `while` statement, reduce the goal to two subgoals.

- One whose conclusion is an HL statement judgement whose precondition is the original goal's precondition, program is the first $n-1$ statements of the original goal's designated program, and postcondition is the negation of the boolean expression of the `while` statement. Actually, the HL statement judgement is universally quantified by a memory of the non-designated program, and references in the precondition to variables of the non-designated program are interpreted in that memory.
- One whose conclusion is a pRHL statement judgement that's the same as that of the original goal except that the `while` statement has been removed.

For example, if the current goal is

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f

pre = true

i <- 0                       (1--)  i <- 0
while (i < 10) {             (2--)  while (i < 10) {
  i <- i + 1                 (2.1)    i <- i + 1
}                            (2--)  }
while (i < 10) {             (3--)
  i <- i + 2                 (3.1)
}                            (3--)

post = ={i}
```

then running

```
rcondf{1} 3.
```

produces the goals

```
Type variables: <none>

forall _, hoare[ i <- 0; while (i < 10) {···} : true ==> ! i < 10]
```

and

```
Type variables: <none>
```

```
&1 (left ) : M.f
&2 (right) : N.f

pre = true

i <- 0                      (1--)  i <- 0
while (i < 10) {            (2--)  while (i < 10) {
  i <- i + 1                (2.1)    i <- i + 1
}                           (2--)  }

post = ={i}
```

`unroll`

Syntax: `unroll` *c*. If the goal's conclusion is an HL statement judgement whose *c*th statement is a `while` statement, then insert before that statement an `if` statement whose boolean expression is the `while` statement's boolean expression, whose `then` part is the `while` statements's body, and whose `else` part is empty.

For example, if the current goal is

```
Type variables: <none>

--------------------------------------------------------------
Context : M.f

pre = true

(1--)  x <- 0
(2--)  z <- 0
(3--)  while (x < y) {
(3.1)    z <- z + x
(3.2)    x <- x + 1
(3--)  }

post = 0 <= z
```

then running

```
unroll 3.
```

produces the goal

```
Type variables: <none>

--------------------------------------------------------------
Context : M.f

pre = true

(1--)  x <- 0
(2--)  z <- 0
(3--)  if (x < y) {
(3.1)    z <- z + x
(3.2)    x <- x + 1
(3--)  }
(4--)  while (x < y) {
(4.1)    z <- z + x
(4.2)    x <- x + 1
(4--)  }

post = 0 <= z
```

And, if the current goal is

```
Type variables: <none>

────────────────────────────────────────────────────────────────────────
Context : M.f

pre = true

(1----)  if (0 <= y) {
(1.1--)    x <- 0
(1.2--)    while (x <> y) {
(1.2.1)      x <- x + 1
(1.2--)    }
(1----)  }

post = 0 <= y
```

then running

```
unroll 1.2.
```

produces the goal

```
Type variables: <none>

────────────────────────────────────────────────────────────────────────
Context : M.f

pre = true

(1----)  if (0 <= y) {
(1.1--)    x <- 0
(1.2--)    if (x <> y) {
(1.2.1)      x <- x + 1
(1.2--)    }
(1.3--)    while (x <> y) {
(1.3.1)      x <- x + 1
(1.3--)    }
(1----)  }

post = 0 <= y
```

Syntax: `unroll{1}` c `|` `unroll{2}` c. If the goal's conclusion is an pRHL statement judgement where the cth statement of the designated program is a `while` statement, then insert before that statement an `if` statement whose boolean expression is the `while` statement's boolean expression, whose `then` part is the `while` statements's body, and whose `else` part is empty.

For example, if the current goal is

```
Type variables: <none>

────────────────────────────────────────────────────────────────────────
&1 (left ) : M.f [programs are in sync]
&2 (right) : M.f

pre = ={y}

(1--)  x <- 0
(2--)  z <- 0
(3--)  while (x < y) {
(3.1)    z <- z + x
```

```
(3.2)    x <- x + 1
(3--)  }

post = ={z}
```

then running

```
unroll{1} 3.
```

produces the goal

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : M.f

pre = ={y}

x <- 0                      (1--)  x <- 0
z <- 0                      (2--)  z <- 0
if (x < y) {                (3--)  while (x < y) {
  z <- z + x                (3.1)    z <- z + x
  x <- x + 1                (3.2)    x <- x + 1
}                           (3--)  }
while (x < y) {             (4--)
  z <- z + x                (4.1)
  x <- x + 1                (4.2)
}                           (4--)

post = ={z}
```

from which running

```
unroll{2} 3.
```

produces the goal

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.f [programs are in sync]
&2 (right) : M.f

pre = ={y}

(1--)  x <- 0
(2--)  z <- 0
(3--)  if (x < y) {
(3.1)    z <- z + x
(3.2)    x <- x + 1
(3--)  }
(4--)  while (x < y) {
(4.1)    z <- z + x
(4.2)    x <- x + 1
(4--)  }

post = ={z}
```

◎ **splitwhile**

Syntax: `splitwhile` c : e. If the goal's conclusion is an HL statement judgement whose cth statement is a `while` statement and e is a well-typed boolean expression in the `while` statement's context, then insert before the `while` statement a copy of the `while` statement in which e is added as a conjunct of the statement's boolean expression.

For example, if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
Context : M.f

pre = true

(1--)  x <- 0
(2--)  z <- 0
(3--)  while (x < y) {
(3.1)    z <- z + x
(3.2)    x <- x + 1
(3--)  }

post = 0 <= z
```

then running

```
splitwhile 3 : z <= 20.
```

produces the goal

```
Type variables: <none>

------------------------------------------------------------------------
Context : M.f

pre = true

(1--)  x <- 0
(2--)  z <- 0
(3--)  while (x < y /\ z <= 20) {
(3.1)    z <- z + x
(3.2)    x <- x + 1
(3--)  }
(4--)  while (x < y) {
(4.1)    z <- z + x
(4.2)    x <- x + 1
(4--)  }

post = 0 <= z
```

Syntax: `splitwhile`{1} c : e | `splitwhile`{2} c : e. If the goal's conclusion is a pRHL statement judgement where the cth statement of the designated program is a `while` statement and e is a well-typed boolean expression in the `while` statement's context, then insert before the `while` statement a copy of the `while` statement in which e is added as a conjunct of the statement's boolean expression.

For example, if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.f [programs are in sync]
&2 (right) : M.f
```

```
pre = ={y}

(1--)  x <- 0
(2--)  z <- 0
(3--)  while (x < y) {
(3.1)    z <- z + x
(3.2)    x <- x + 1
(3--)  }

post = ={z}
```

then running

```
splitwhile{2} 3 : z <= 20.
```

produces the goal

```
Type variables: <none>

-----------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : M.f

pre = ={y}

x <- 0                      (1--)  x <- 0
z <- 0                      (2--)  z <- 0
while (x < y) {             (3--)  while (x < y /\ z <= 20) {
  z <- z + x                (3.1)    z <- z + x
  x <- x + 1                (3.2)    x <- x + 1
}                           (3--)  }
                            (4--)  while (x < y) {
                            (4.1)    z <- z + x
                            (4.2)    x <- x + 1
                            (4--)  }

post = ={z}
```

from which running

```
splitwhile{1} 3 : z <= 20.
```

produces the goal

```
Type variables: <none>

-----------------------------------------------------------------------
&1 (left ) : M.f [programs are in sync]
&2 (right) : M.f

pre = ={y}

(1--)  x <- 0
(2--)  z <- 0
(3--)  while (x < y /\ z <= 20) {
(3.1)    z <- z + x
(3.2)    x <- x + 1
(3--)  }
(4--)  while (x < y) {
(4.1)    z <- z + x
(4.2)    x <- x + 1
```

```
(4--)  }

post = ={z}
```

fission

Syntax: **fission** $c!l$ @ m, n. HL statement judgement version. Fails unless $0 \leq l$ and $0 \leq m \leq n$ and the cth statement of the program is a **while** statement, and there are at least l statements right before the **while** statement, at its level, and the body of the **while** statement has at least n statements.

Let

- s_1 be the l statements before the **while** statement at position c;
- e be the boolean expression of the **while** statement;
- s_2 be the first m statements of the body of the **while** statement;
- s_3 be the next $n - m$ statements of the body of the **while** statement;
- s_4 be the rest of the body of the **while** statement.

Fails unless:

- e doesn't reference the variables written by s_2 and s_3;
- s_1 and s_4 don't read or write the variables written by s_2 and s_3;
- s_2 and s_3 don't write the variables written by s_1 and s_4;
- s_2 and s_3 don't read or write the variables written by the other.

The tactic replaces

s_1 **while** (e) { s_2 s_3 s_4 }

by

s_1 **while** (e) { s_2 s_4 }
s_1 **while** (e) { s_3 s_4 }

For example, if the current goal is

```
Type variables: <none>

Context : M.f

pre = 0 < n

(1--)  x <- 0
(2--)  y <- 0
(3--)  i <- 0
(4--)  j <- 0
(5--)  while (i + j < n) {
(5.1)    x <- x * i
(5.2)    x <- x + (j + 1)
(5.3)    y <- y * j
(5.4)    y <- y + (i + 2)
(5.5)    i <- i + 1
(5.6)    j <- j + 2
(5--)  }

post = 0 < x + y
```

then running

```
fission 5!2 @ 2, 4.
```

produces the goal

```
Type variables: <none>

------------------------------------------------------------------------
Context : M.f

pre = 0 < n

(1--)  x <- 0
(2--)  y <- 0
(3--)  i <- 0
(4--)  j <- 0
(5--)  while (i + j < n) {
(5.1)    x <- x * i
(5.2)    x <- x + (j + 1)
(5.3)    i <- i + 1
(5.4)    j <- j + 2
(5--)  }
(6--)  i <- 0
(7--)  j <- 0
(8--)  while (i + j < n) {
(8.1)    y <- y * j
(8.2)    y <- y + (i + 2)
(8.3)    i <- i + 1
(8.4)    j <- j + 2
(8--)  }

post = 0 < x + y
```

Syntax: `fission` c `@` m`,` n. Equivalent to `fission` c`!1` `@` m`,` n.

Syntax: `fission{1}` $\cdots$ `|` `fission{2}` $\cdots$. The pRHL versions of the above variants, working on the designated program.

fusion

Syntax: `fusion` c`!`l `@` m`,` n. HL statement judgement version. Fails unless $0 \leq l$ and $0 \leq m$ and $0 \leq n$ and the cth statement of the program is a `while` statement, and there are at least l statements right before the `while` statement, at its level, and the part of the program beginning from the l statements before the while loop may be uniquely matched against

```
s1 while (e) { s2 s4 }
s1 while (e) { s3 s4 }
```

where:

- s_1 has length l;
- s_2 has length m;
- s_3 has length n;
- e doesn't reference the variables written by s_2 and s_3;
- s_1 and s_4 don't read or write the variables written by s_2 and s_3;
- s_2 and s_3 don't write the variables written by s_1 and s_4;

- s_2 and s_3 don't read or write the variables written by the other.

The tactic replaces

```
s1 while (e) { s2 s4 }
s1 while (e) { s3 s4 }
```

by

```
s1 while (e) { s2 s3 s4 }
```

For example, if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
Context : M.f

pre = 0 < n

(1--)  x <- 0
(2--)  y <- 0
(3--)  i <- 0
(4--)  j <- 0
(5--)  while (i + j < n) {
(5.1)    x <- x * i
(5.2)    x <- x + (j + 1)
(5.3)    i <- i + 1
(5.4)    j <- j + 2
(5--)  }
(6--)  i <- 0
(7--)  j <- 0
(8--)  while (i + j < n) {
(8.1)    y <- y * j
(8.2)    y <- y + (i + 2)
(8.3)    i <- i + 1
(8.4)    j <- j + 2
(8--)  }

post = 0 < x + y
```

then running

```
fusion 5!2 @ 2, 2.
```

produces the goal

```
Type variables: <none>

------------------------------------------------------------------------
Context : M.f

pre = 0 < n

(1--)  x <- 0
(2--)  y <- 0
(3--)  i <- 0
(4--)  j <- 0
(5--)  while (i + j < n) {
(5.1)    x <- x * i
(5.2)    x <- x + (j + 1)
(5.3)    y <- y * j
```

```
(5.4)     y <- y + (i + 2)
(5.5)     i <- i + 1
(5.6)     j <- j + 2
(5--)   }

post = 0 < x + y
```

Syntax: `fusion` c `@` m`,` n. Equivalent to `fusion` c`!1 @` m`,` n.

Syntax: `fusion{1}` $\cdots$ `|` `fusion{2}` $\cdots$. The pRHL versions of the above variants, working on the designated program.

`alias`

Syntax: `alias` c `with` x. If the goal's conclusion is an HL statement judgement whose program's cth statement is an assignment statement, and x is an identifier, then replace the assignment statement by the following two statements:

- an assignment statement of the same kind as the original assignment statement (ordinary, random, procedure call) whose left-hand side is x, and whose right-hand side is the right-hand side of the original assignment statement;
- an ordinary assignment statement whose left-hand side is the left-hand side of the original assignment statement, and whose right-hand side is x.

If x is a local variable of the program, a fresh name is generated by adding digits to the end of x.

Syntax: `alias` c. Equivalent to `alias` c `with x`.

Syntax: `alias` c x `=` e. If the program has an cth statement, and the expression e is well-typed in the context of the program, insert before the cth statement an ordinary assignment statement whose left-hand side is x and whose right-hand side is e. If x is a local variable of the program, a fresh name is generated by adding digits to the end of x.

Syntax: `alias{1}` $\cdots$ `|` `alias{2}` $\cdots$. The pRHL versions of the preceding forms, where the aliasing is done in the designated program.

For example, if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
Context : M.f

pre = true

(1)  x <- 10
(2)  (y, z) <- (x + 1, 6)

post = y + z = 17
```

then running

```
alias 2 with w.
```

produces the goal

```
Type variables: <none>

------------------------------------------------------------------------
Context : M.f
```

```
pre = true

(1)  x <- 10
(2)  w <- (x + 1, 6)
(3)  (y, z) <- w

post = y + z = 17
```

from which running

```
alias 3 u = w.`1 + 7.
```

produces the goal

```
Type variables: <none>

------------------------------------------------------------------------
Context : M.f

pre = true

(1)  x <- 10
(2)  w <- (x + 1, 6)
(3)  u <- w.`1 + 7
(4)  (y, z) <- w

post = y + z = 17
```

from which running

```
alias 3.
```

produces the goal

```
Type variables: <none>

------------------------------------------------------------------------
Context : M.f

pre = true

(1)  x <- 10
(2)  w <- (x + 1, 6)
(3)  x0 <- w.`1 + 7
(4)  u <- x0
(5)  (y, z) <- w

post = y + z = 17
```

cfold

Syntax: `cfold` c `!` m. Fails unless $m \geq 0$. If the goal's conclusion is an HL statement judgement in which statement c of the judgement's program is an ordinary assignment statement in which constant values are assigned to local identifiers, and the following statement block of length m does not write any of those identifiers, then replace all occurrences of the assigned identifiers in that statement block by the constants assigned to them, and move the assignment statement to after the modified statement block.

For example, if the current goal is

```
Type variables: <none>
```

```
Context : M.f

pre = true

(1)  x <- 1
(2)  y <- x + 1
(3)  z <- y + x + 2
(4)  w <- x - z

post = w = -4
```

then running

```
cfold 1 ! 1.
```

produces the goal

```
Type variables: <none>

Context : M.f

pre = true

(1)  y <- 1 + 1
(2)  x <- 1
(3)  z <- y + x + 2
(4)  w <- x - z

post = w = -4
```

from which running

```
cfold 2.
```

produces the goal

```
Type variables: <none>

Context : M.f

pre = true

(1)  y <- 1 + 1
(2)  z <- y + 1 + 2
(3)  w <- 1 - z
(4)  x <- 1

post = w = -4
```

from which running

```
cfold 1.
```

produces the goal

```
Type variables: <none>

Context : M.f
```

```
pre = true

(1)  z <- 1 + 1 + 1 + 2
(2)  w <- 1 - z
(3)  x <- 1
(4)  y <- 1 + 1

post = w = -4
```

from which running

```
cfold 1.
```

produces the goal

```
Type variables: <none>

----------------------------------------------------------------
Context : M.f

pre = true

(1)  w <- 1 - (1 + 1 + 1 + 2)
(2)  x <- 1
(3)  y <- 1 + 1
(4)  z <- 1 + 1 + 1 + 2

post = w = -4
```

Syntax: `cfold{1}` c `!` m `|` `cfold{2}` c `!` m. Like the HL version, but operating on the designed program of a pRHL judgement's conclusion.

Syntax: `cfold{1}` c `|` `cfold{2}` c. Like the general cases, but where m is set so as to be the number of statements after the assignment statement.

`kill`

Syntax: `kill` c `!` m. Fails unless $m \geq 0$. If the goal's conclusion is an HL statement judgement whose program has a statement block starting at position c and having length m (when $m = 0$, this block is empty), and the variables written by this statement block aren't used in the judgement's postcondition or read by the rest of the program, then reduce the goal to two subgoals.

- One whose conclusion is a pHL statement judgement whose pre- and postconditions are `true`, whose program is the statement block, and whose bound part is `= 1%r`.
- One that's identical to the original goal except that the statement block has been removed.

For example, if the current goal is

```
Type variables: <none>

----------------------------------------------------------------
Context : M.f

pre = M.y = 3

(1)  M.y <- M.y + 1
(2)  M.x <- 0
(3)  M.x <- M.x + 1
(4)  M.y <- M.y + 1

post = M.y = 5
```

then running

```
kill 2 ! 2.
```

produces the goals

```
Type variables: <none>

Context : M.f
Bound   : [=] 1%r

pre = true

(1)  M.x <- 0
(2)  M.x <- M.x + 1

post = true
```

and

```
Type variables: <none>

Context : M.f

pre = M.y = 3

(1)  M.y <- M.y + 1
(2)  M.y <- M.y + 1

post = M.y = 5
```

Syntax: `kill{1}` c `!` m `|` `kill{2}` c `!` m. Like the HL case but for pRHL judgements, where the statement block to be killed is in the designated program.

For example, if the current goal is

```
Type variables: <none>

&1 (left ) : M.f [programs are in sync]
&2 (right) : M.f

pre = ={M.y} /\ M.y{1} = 3

(1)  M.y <- M.y + 1
(2)  M.x <- 0
(3)  M.x <- M.x + 1
(4)  M.y <- M.y + 1

post = ={M.y} /\ M.y{1} = 5
```

then running

```
kill{2} 2 ! 2.
```

produces the goals

```
Type variables: <none>

Context : M.f
```

```
Bound   : [=] 1%r

pre = true

(1)  M.x <- 0
(2)  M.x <- M.x + 1

post = true
```

and

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : M.f

pre = ={M.y} /\ M.y{1} = 3

M.y <- M.y + 1             (1)  M.y <- M.y + 1
M.x <- 0                   (2)  M.y <- M.y + 1
M.x <- M.x + 1             (3)
M.y <- M.y + 1             (4)

post = ={M.y} /\ M.y{1} = 5
```

Syntax: `kill` c | `kill{1}` c | `kill{2}` c. Like the general cases, but with $m = 1$.

Syntax: `kill` c `! *` | `kill{1}` c `! *` | `kill{2}` c `! *`. Like the general cases, but with m set so that the statement block to be killed is the rest of the current level of the (designated) program.

3.4.3 Tactics for Reasoning about Specifications

`symmetry`

Syntax: `symmetry`. If the goal's conclusion is a pRHL (statement) judgement, swap the two programs, transforming the pre- and postconditions by swapping the memories they refer to.

For example, if the current goal is

```
Type variables: <none>

pre = x{1} = y{2} /\ x{2} = - y{1}

    M.f ~ N.g

post = 1 - res{1} = res{2}
```

then running

```
symmetry.
```

produces the goal

```
Type variables: <none>

pre = x{2} = y{1} /\ x{1} = - y{2}

    N.g ~ M.f

post = 1 - res{2} = res{1}
```

And, if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : N.g

pre =
  (x{1}, y{1}).`1 = (x{2}, y{2}).`2 /\
  (x{2}, y{2}).`1 = - (x{1}, y{1}).`2

z <- x - y                (1)  z <- x + y

post = 1 - (z{1} + 1) = -z{2}
```

then running

```
symmetry.
```

produces the goal

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : N.g
&2 (right) : M.f

pre =
  (x{2}, y{2}).`1 = (x{1}, y{1}).`2 /\
  (x{1}, y{1}).`1 = - (x{2}, y{2}).`2

z <- x + y                (1)  z <- x - y

post = 1 - (z{2} + 1) = -z{1}
```

.

`transitivity`

Syntax: `transitivity` $N.r$ `(`P_1 `==>` Q_1`)` `(`P_2 `==>` Q_2`)`. Reduces a goal whose conclusion is a pRHL judgement (*not* statement judgement)

$$\texttt{equiv}[M_1.p_1 \sim M_2.p_2 \ : P \ \texttt{==>}\ Q]$$

to goals whose conclusions are

- `equiv`$[M_1.p_1 \sim N.r \ : P_1 \ \texttt{==>}\ Q_1]$ and
- `equiv`$[N.r \sim M_2.p_2 \ : P_2 \ \texttt{==>}\ Q_2]$,

preceded by two auxiliary goals. The tactic fails if the P_i and Q_i aren't compatible with these left and right programs. The conclusion of the first auxiliary goal checks that P implies the conjunction of P_1 and P_2, where each corresponding pair of `&2` memory references in P_1 and `&1` reference in P_2 is existentially quantified. And the conclusion of the second auxilary goal checks that the conjuction of Q_1 and Q_2 implies Q, where each corresponding pair of `&2` references in Q_1 and `&1` references in Q_2 are universally quantified.

For example, consider the modules

```
module M = {
  proc f(n : int, m : int) : int = {
    var i, x : int;
    i <- 0; x <- 0;
```

```
    while (i < n) {
      x <- x + m; i <- i + 1;
    }
    return x;
  }
}.
module N = {
  proc g(n : int, m : int) : int = {
    var j, y : int;
    j <- 0; y <- 0;
    while (j < n) {
      y <- y + m; j <- j + 1;
    }
    return y;
  }
}.
module R = {
  proc h(n : int, m : int) : int = {
    return n * m;
  }
}.
```

If the current goal is

```
Type variables: <none>

pre = 0 <= n{1} /\ 0 <= n{2} /\ n{1} * m{1} = n{2} * m{2}

    M.f ~ N.g

post = ={res}
```

then running

```
transitivity
     R.h
     (0 <= n{1} /\ (n * m){1} = (n * m){2} ==> ={res})
     (0 <= n{2} /\ (n * m){1} = (n * m){2} ==> ={res}).
```

produces the four goals

```
Type variables: <none>

forall &1 &2,
  0 <= n{1} /\ 0 <= n{2} /\ n{1} * m{1} = n{2} * m{2} =>
  exists (arg : int * int),
    (0 <= n{1} /\ n{1} * m{1} = arg.`1 * arg.`2) /\
    0 <= n{2} /\ arg.`1 * arg.`2 = n{2} * m{2}
```

and

```
Type variables: <none>

forall &1 &m &2, res{1} = res{m} => res{m} = res{2} => ={res}
```

and

```
Type variables: <none>
```

```
pre = 0 <= n{1} /\ n{1} * m{1} = n{2} * m{2}

    M.f ~ R.h

post = ={res}
```

and

```
Type variables: <none>

pre = 0 <= n{2} /\ n{1} * m{1} = n{2} * m{2}

    R.h ~ N.g

post = ={res}
```

Syntax: `transitivity{`i`} {`s`} (`P_1` ==> `Q_1`) (`P_2` ==> `Q_2`)`.
Reduces a goal whose conclusion is a pRHL *statement* judgement with precondition P, postcondition Q, left program (statement sequence) s_1 and right program s_2 to goals whose conclusions are pRHL statement judgements:

- `equiv[`s_1 `~` s `:` P_1 `==>` Q_1`]` and
- `equiv[`s `~` s_2 `:` P_2 `==>` Q_2`]`,

preceded by two auxiliary goals. If the side $i = 1$, then the statement sequence s may only use variables and unqualified procedures of the left program; when $i = 2$, it may only use variables and unqualified procedures of the right program. The tactic fails if the P_i and Q_i aren't compatible with these left and right programs. The conclusion of the first auxiliary goal checks that P implies the conjunction of P_1 and P_2, where each corresponding pair of `&2` memory references in P_1 and `&1` reference in P_2 is existentially quantified. And the conclusion of the second auxilary goal checks that the conjuction of Q_1 and Q_2 implies Q, where each corresponding pair of `&2` references in Q_1 and `&1` references in Q_2 are universally quantified.

Consider the modules `M`, `N` and `R` of the preceding case. If the current goal is

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.g

pre =
  j{2} = 0 /\
  y{2} = 0 /\
  i{1} = 0 /\
  x{1} = 0 /\ 0 <= n{1} /\ 0 <= n{2} /\ n{1} * m{1} = n{2} * m{2}

while (i < n) {             (1--)  while (j < n) {
  x <- x + m                (1.1)    y <- y + m
  i <- i + 1                (1.2)    j <- j + 1
}                           (1--)  }

post = x{1} = y{2}
```

then running

```
transitivity{1}
     {x <- n * m;}
     (0 <= n{1} /\ i{1} = 0 /\ x{1} = 0 /\ (n * m){1} = (n * m){2} ==>
```

```
 ={x})
(0 <= n{2} /\ j{2} = 0 /\ y{2} = 0 /\ (n * m){1} = (n * m){2} ==>
 x{1} = y{2}).
```

produces the four goals

```
Type variables: <none>

-----------------------------------------------------------------------
forall &1 &2,
  j{2} = 0 /\
  y{2} = 0 /\
  i{1} = 0 /\
  x{1} = 0 /\ 0 <= n{1} /\ 0 <= n{2} /\ n{1} * m{1} = n{2} * m{2} =>
  exists (m n : int),
    (0 <= n{1} /\ i{1} = 0 /\ x{1} = 0 /\ n{1} * m{1} = n * m) /\
    0 <= n{2} /\ j{2} = 0 /\ y{2} = 0 /\ n * m = n{2} * m{2}
```

and

```
Type variables: <none>

-----------------------------------------------------------------------
forall &1 &m &2, x{1} = x{m} => x{m} = y{2} => x{1} = y{2}
```

and

```
Type variables: <none>

-----------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : M.f

pre =
  0 <= n{1} /\ i{1} = 0 /\ x{1} = 0 /\ n{1} * m{1} = n{2} * m{2}

while (i < n) {             (1--)  x <- n * m
  x <- x + m                (1.1)
  i <- i + 1                (1.2)
}                           (1--)

post = ={x}
```

and

```
Type variables: <none>

-----------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : N.g

pre =
  0 <= n{2} /\ j{2} = 0 /\ y{2} = 0 /\ n{1} * m{1} = n{2} * m{2}

x <- n * m                  (1--)  while (j < n) {
                            (1.1)    y <- y + m
                            (1.2)    j <- j + 1
                            (1--)  }

post = x{1} = y{2}
```

And, if the current goal is

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.g

pre =
  j{2} = 0 /\
  y{2} = 0 /\
  i{1} = 0 /\
  x{1} = 0 /\ 0 <= n{1} /\ 0 <= n{2} /\ n{1} * m{1} = n{2} * m{2}

while (i < n) {             (1--)  while (j < n) {
  x <- x + m                (1.1)    y <- y + m
  i <- i + 1                (1.2)    j <- j + 1
}                           (1--)  }

post = x{1} = y{2}
```

then running

```
transitivity{2}
     {y <- n * m;}
     (0 <= n{1} /\ i{1} = 0 /\ x{1} = 0 /\ (n * m){1} = (n * m){2} ==>
      x{1} = y{2})
     (0 <= n{2} /\ j{2} = 0 /\ y{2} = 0 /\ (n * m){1} = (n * m){2} ==>
      ={y}).
```

produces the four goals

```
Type variables: <none>

forall &1 &2,
  j{2} = 0 /\
  y{2} = 0 /\
  i{1} = 0 /\
  x{1} = 0 /\ 0 <= n{1} /\ 0 <= n{2} /\ n{1} * m{1} = n{2} * m{2} =>
  exists (m n : int),
    (0 <= n{1} /\ i{1} = 0 /\ x{1} = 0 /\ n{1} * m{1} = n * m) /\
    0 <= n{2} /\ j{2} = 0 /\ y{2} = 0 /\ n * m = n{2} * m{2}
```

and

```
Type variables: <none>

forall &1 &m &2, x{1} = y{m} => y{m} = y{2} => x{1} = y{2}
```

and

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.g

pre =
  0 <= n{1} /\ i{1} = 0 /\ x{1} = 0 /\ n{1} * m{1} = n{2} * m{2}

while (i < n) {             (1--)  y <- n * m
```

```
  x <- x + m               (1.1)
  i <- i + 1               (1.2)
}                          (1--)

post = x{1} = y{2}
```

and

```
Type variables: <none>

&1 (left ) : N.g
&2 (right) : N.g

pre =
  0 <= n{2} /\ j{2} = 0 /\ y{2} = 0 /\ n{1} * m{1} = n{2} * m{2}

y <- n * m                 (1--)  while (j < n) {
                           (1.1)    y <- y + m
                           (1.2)    j <- j + 1
                           (1--)  }

post = ={y}
```

conseq

Syntax: `conseq (_ : P ==> Q)`.
If the goal's conclusion is a PRHL or HL judgement or statement judgement, weaken the conclusion's precondition to P, and strengthen its postcondition to Q, generating initial auxilary subgoals checking that this reduction is sound. Fails if P and Q aren't compatible with the judgement type. The conclusion of the first auxiliary subgoal checks that the precondition P' of the original goal's conclusion implies P. The conclusion of the second auxiliary subgoal checks that Q implies the postcondition Q' of the original goal's conclusion, except that any memory references to variables that may be modified by the conclusion's program(s) are universally quantified in Q and Q', and P is also included as an assumption.
P or Q may be replaced by _, in which case the pre- or postcondition is left unchanged. When a pre- or postcondition is unchanged, the corresponding auxilary subgoal isn't generated (as its proof would be trivial). When the goal's conclusion is a judgement (not a statement judgement), a proof term whose conclusion is a judgement on the procedure(s) of the original goals's conclusion may be supplied as the argument to `conseq`, in which case P and Q are taken to be the pre- and postconditions of this judgement, and only the auxilary subgoals are generated.

For example, if the current goal is

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f

pre =
  ={z} /\
  M.x{1} = N.x{2} /\
  M.y{1} = N.y{2} /\ z{1} < M.x{1} /\ z{1} < M.y{1}

if (z <= M.x \/ z <= M.y) {  (1--)  if (z <= N.x) {
  M.x <- M.x + 1             (1.1)    N.x <- N.x - 1
}                            (1--)  }

post = M.y{1} = N.y{2} /\ `|M.x{1} - N.x{2}| <= 2
```

then running

```
conseq (_ :
          ={z} /\ M.x{1} = N.x{2} /\ M.y{1} = N.y{2} /\
          z{1} <= M.x{1} /\ z{1} <= M.y{1} ==> _).
```

produces the goals

```
Type variables: <none>

------------------------------------------------------------------------
forall &1 &2,
  ={z} /\
  M.x{1} = N.x{2} /\
  M.y{1} = N.y{2} /\ z{1} < M.x{1} /\ z{1} < M.y{1} =>
  ={z} /\
  M.x{1} = N.x{2} /\
  M.y{1} = N.y{2} /\ z{1} <= M.x{1} /\ z{1} <= M.y{1}
```

and

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : N.f

pre =
  ={z} /\
  M.x{1} = N.x{2} /\
  M.y{1} = N.y{2} /\ z{1} <= M.x{1} /\ z{1} <= M.y{1}

if (z <= M.x \/ z <= M.y) {  (1--)  if (z <= N.x) {
  M.x <- M.x + 1             (1.1)    N.x <- N.x - 1
}                            (1--)  }

post = M.y{1} = N.y{2} /\ `|M.x{1} - N.x{2}| <= 2
```

Continuing from the last of these goals, running

```
conseq (_ : _ ==> `|M.x{1} - N.x{2}| <= 2).
```

produces the goals

```
Type variables: <none>

------------------------------------------------------------------------
forall &1 &2,
  ={z} /\
  M.x{1} = N.x{2} /\ M.y{1} = N.y{2} /\ z{1} <= M.x{1} /\ z{1} <= M.y{1} =>
  forall (x_L x_R : int),
    `|x_L - x_R| <= 2 => M.y{1} = N.y{2} /\ `|x_L - x_R| <= 2
```

and

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : N.f

pre =
  ={z} /\
```

```
  M.x{1} = N.x{2} /\ M.y{1} = N.y{2} /\ z{1} <= M.x{1} /\ z{1} <= M.y{1}

if (z <= M.x \/ z <= M.y) {  (1--)  if (z <= N.x) {
  M.x <- M.x + 1             (1.1)    N.x <- N.x - 1
}                            (1--)  }

post = `|M.x{1} - N.x{2}| <= 2
```

Continuing from the last of these goals, running

```
conseq (_ :
           ={z} /\ M.x{1} = N.x{2} /\ z{1} <= N.x{2} ==>
           M.x{1} = N.x{2} \/ M.x{1} = N.x{2} + 2).
```

produces the goals

```
Type variables: <none>

forall &1 &2,
  ={z} /\
  M.x{1} = N.x{2} /\
  M.y{1} = N.y{2} /\ z{1} <= M.x{1} /\ z{1} <= M.y{1} =>
  ={z} /\ M.x{1} = N.x{2} /\ z{1} <= N.x{2}
```

and

```
Type variables: <none>

forall &1 &2,
  ={z} /\
  M.x{1} = N.x{2} /\
  M.y{1} = N.y{2} /\ z{1} <= M.x{1} /\ z{1} <= M.y{1} =>
  forall (x_L x_R : int),
    x_L = x_R \/ x_L = x_R + 2 => `|x_L - x_R| <= 2
```

and

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f

pre = ={z} /\ M.x{1} = N.x{2} /\ z{1} <= N.x{2}

if (z <= M.x \/ z <= M.y) {  (1--)  if (z <= N.x) {
  M.x <- M.x + 1             (1.1)    N.x <- N.x - 1
}                            (1--)  }

post = M.x{1} = N.x{2} \/ M.x{1} = N.x{2} + 2
```

If the current goal is

```
Type variables: <none>

pre =
  ={z} /\
  M.x{1} = N.x{2} /\
  M.y{1} = N.y{2} /\ z{1} < M.x{1} /\ z{1} < M.y{1}
```

```
    M.f ~ N.f

post = M.y{1} = N.y{2} /\ `|M.x{1} - N.x{2}| <= 2
```

then running

```
conseq (_ :
           ={z} /\ M.x{1} = N.x{2} /\ M.y{1} = N.y{2} /\
           z{1} <= M.x{1} /\ z{1} <= M.y{1} ==> _).
```

produces the goals

```
Type variables: <none>

--------------------------------------------------------------------
forall &1 &2,
  ={z} /\
  M.x{1} = N.x{2} /\
  M.y{1} = N.y{2} /\ z{1} < M.x{1} /\ z{1} < M.y{1} =>
  ={z} /\
  M.x{1} = N.x{2} /\
  M.y{1} = N.y{2} /\ z{1} <= M.x{1} /\ z{1} <= M.y{1}
```

and

```
Type variables: <none>

--------------------------------------------------------------------
pre =
  ={z} /\
  M.x{1} = N.x{2} /\
  M.y{1} = N.y{2} /\ z{1} <= M.x{1} /\ z{1} <= M.y{1}

    M.f ~ N.f

post = M.y{1} = N.y{2} /\ `|M.x{1} - N.x{2}| <= 2
```

Continuing from the last of these goals, running

```
conseq (_ : _ ==> `|M.x{1} - N.x{2}| <= 2).
```

produces the goals

```
Type variables: <none>

--------------------------------------------------------------------
forall &1 &2,
  ={z} /\
  M.x{1} = N.x{2} /\ M.y{1} = N.y{2} /\ z{1} <= M.x{1} /\ z{1} <= M.y{1} =>
  forall (x_L x_R : int),
    `|x_L - x_R| <= 2 => M.y{1} = N.y{2} /\ `|x_L - x_R| <= 2
```

and

```
Type variables: <none>

--------------------------------------------------------------------
pre =
  ={z} /\
  M.x{1} = N.x{2} /\ M.y{1} = N.y{2} /\ z{1} <= M.x{1} /\ z{1} <= M.y{1}

    M.f ~ N.f
```

```
post = `|M.x{1} - N.x{2}| <= 2
```

Continuing from the last of these goals, running

```
conseq (_ :
          ={z} /\ M.x{1} = N.x{2} /\ z{1} <= N.x{2} ==>
          M.x{1} = N.x{2} \/ M.x{1} = N.x{2} + 2).
```

produces the goals

```
Type variables: <none>

------------------------------------------------------------------------
forall &1 &2,
  ={z} /\
  M.x{1} = N.x{2} /\
  M.y{1} = N.y{2} /\ z{1} <= M.x{1} /\ z{1} <= M.y{1} =>
  ={z} /\ M.x{1} = N.x{2} /\ z{1} <= N.x{2}
```

and

```
Type variables: <none>

------------------------------------------------------------------------
forall &1 &2,
  ={z} /\
  M.x{1} = N.x{2} /\
  M.y{1} = N.y{2} /\ z{1} <= M.x{1} /\ z{1} <= M.y{1} =>
  forall (x_L x_R : int),
    x_L = x_R \/ x_L = x_R + 2 => `|x_L - x_R| <= 2
```

and

```
Type variables: <none>

------------------------------------------------------------------------
pre = ={z} /\ M.x{1} = N.x{2} /\ z{1} <= N.x{2}

    M.f ~ N.f

post = M.x{1} = N.x{2} \/ M.x{1} = N.x{2} + 2
```

Given lemma

```
lemma M_N_3 :
  equiv[M.f ~ N.f :
        ={z} /\ M.x{1} = N.x{2} /\ z{1} <= N.x{2} ==>
        M.x{1} = N.x{2} \/ M.x{1} = N.x{2} + 2].
```

if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
pre =
  ={z} /\
  M.x{1} = N.x{2} /\
  M.y{1} = N.y{2} /\ z{1} < M.x{1} /\ z{1} < M.y{1}

    M.f ~ N.f

post = M.y{1} = N.y{2} /\ `|M.x{1} - N.x{2}| <= 2
```

then running

```
conseq M_N_3.
```

produces the goals

```
Type variables: <none>

------------------------------------------------------------------------
forall &1 &2,
  ={z} /\
  M.x{1} = N.x{2} /\
  M.y{1} = N.y{2} /\ z{1} < M.x{1} /\ z{1} < M.y{1} =>
  ={z} /\ M.x{1} = N.x{2} /\ z{1} <= N.x{2}
```

and

```
Type variables: <none>

------------------------------------------------------------------------
forall &1 &2,
  ={z} /\
  M.x{1} = N.x{2} /\
  M.y{1} = N.y{2} /\ z{1} < M.x{1} /\ z{1} < M.y{1} =>
  forall (x_L x_R : int),
    x_L = x_R \/ x_L = x_R + 2 =>
    M.y{1} = N.y{2} /\ `|x_L - x_R| <= 2
```

If the current goal is

```
Type variables: <none>

x': int
------------------------------------------------------------------------
Context : M.f

pre = x' = M.x /\ z < M.x /\ z < M.y

(1--)  if (z <= M.x \/ z <= M.y) {
(1.1)    M.x <- M.x + 1
(1--)  }

post = x' < M.x
```

then running

```
conseq (_ : x' = M.x /\ z <= M.y ==> x' = M.x - 1).
```

produces the goals

```
Type variables: <none>

x': int
------------------------------------------------------------------------
forall &hr,
  x' = M.x{hr} /\ z{hr} < M.x{hr} /\ z{hr} < M.y{hr} =>
  x' = M.x{hr} /\ z{hr} <= M.y{hr}
```

and

```
Type variables: <none>

x': int
```

```
forall &hr,
  x' = M.x{hr} /\ z{hr} < M.x{hr} /\ z{hr} < M.y{hr} =>
  forall (x : int), x' = x - 1 => x' < x
```

and

```
Type variables: <none>

x': int
------------------------------------------------------------------------
Context : M.f

pre = x' = M.x /\ z <= M.y

(1--)  if (z <= M.x \/ z <= M.y) {
(1.1)    M.x <- M.x + 1
(1--)  }

post = x' = M.x - 1
```

If the current goal is

```
Type variables: <none>

x': int
------------------------------------------------------------------------
pre = x' = M.x /\ z < M.x /\ z < M.y

    M.f

post = x' < M.x
```

then running

```
conseq (_ : x' = M.x /\ z <= M.y ==> x' = M.x - 1).
```

produces the goals

```
Type variables: <none>

x': int
------------------------------------------------------------------------
forall &hr,
  x' = M.x{hr} /\ z{hr} < M.x{hr} /\ z{hr} < M.y{hr} =>
  x' = M.x{hr} /\ z{hr} <= M.y{hr}
```

and

```
Type variables: <none>

x': int
------------------------------------------------------------------------
forall &hr,
  x' = M.x{hr} /\ z{hr} < M.x{hr} /\ z{hr} < M.y{hr} =>
  forall (x : int), x' = x - 1 => x' < x
```

and

```
Type variables: <none>

x': int
------------------------------------------------------------------------
```

```
pre = x' = M.x /\ z <= M.y

    M.f

post = x' = M.x - 1
```

Given lemma

```
lemma M_3 (x' : int) :
  hoare[M.f : x' = M.x /\ z <= M.y ==> x' = M.x - 1].
```

if the current goal is

```
Type variables: <none>

x': int
------------------------------------------------------------------------
pre = x' = M.x /\ z < M.x /\ z < M.y

    M.f

post = x' < M.x
```

then running

```
conseq (M_3 x').
```

produces the goals

```
Type variables: <none>

x': int
------------------------------------------------------------------------
forall &hr,
  x' = M.x{hr} /\ z{hr} < M.x{hr} /\ z{hr} < M.y{hr} =>
  x' = M.x{hr} /\ z{hr} <= M.y{hr}
```

and

```
Type variables: <none>

x': int
------------------------------------------------------------------------
forall &hr,
  x' = M.x{hr} /\ z{hr} < M.x{hr} /\ z{hr} < M.y{hr} =>
  forall (x : int), x' = x - 1 => x' < x
```

Syntax: `conseq (_ :` P `==>` Q`) (_ :` P_1 `==>` Q_1`) (_ :` P_2 `==>` Q_2`).`
This form only applies to pRHL judgements and statement judgements, reducing the goal's conclusion to

- an HL (statement) judgement for the left program with precondition P_1 and postcondition Q_1;
- an HL (statement) judgement for the right program with precondition P_2 and postcondition Q_2; and
- a pRHL (statement) judgement whose precondition is P, postcondition is Q, and programs are as in the original judgement;

As before, auxiliary goals are generated whose conclusions check the validity of the reduction: that the original conclusion's precondition P' implies the conjunction of P, P_1 and P_2; and that the conjunction of Q, Q_1 and Q_2 implies the original conclusion's postcondition Q'. One of the HL specifications may be replaced by `_`, which is equivalent to `conseq (_ : true ==> true)`. And in the case of a pRHL judgement (not a statement judgement), proof terms may be used as the arguments to `conseq`.

For example, if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : N.f

pre = ={b} /\ M.x{1} %% 2 + N.y{2} %% 2 = 0

M.x <- M.x + 2              (1)  N.y <- N.y - 2
b' <$ {0,1}                 (2)  b' <$ {0,1}
b <- b /\ b'                (3)  b <- b /\ b'

post = ={b} /\ M.x{1} %% 2 + N.y{2} %% 2 = 0
```

then running

```
conseq (_ : ={b} ==> ={b})
          (_ : M.x %% 2 = 0 ==> M.x %% 2 = 0)
          (_ : N.y %% 2 = 0 ==> N.y %% 2 = 0).
```

produces the goals

```
Type variables: <none>

------------------------------------------------------------------------
forall &1 &2,
  ={b} /\ M.x{1} %% 2 + N.y{2} %% 2 = 0 =>
  ={b} /\ M.x{1} %% 2 = 0 /\ N.y{2} %% 2 = 0
```

and

```
Type variables: <none>

------------------------------------------------------------------------
forall &1 &2,
  ={b} /\ M.x{1} %% 2 + N.y{2} %% 2 = 0 =>
  forall (x_L : int) (b_L : bool) (y_R : int) (b_R : bool),
    b_L = b_R /\ x_L %% 2 = 0 /\ y_R %% 2 = 0 =>
    b_L = b_R /\ x_L %% 2 + y_R %% 2 = 0
```

and

```
Type variables: <none>

------------------------------------------------------------------------
Context : M.f

pre = M.x %% 2 = 0

(1)  M.x <- M.x + 2
(2)  b' <$ {0,1}
(3)  b <- b /\ b'

post = M.x %% 2 = 0
```

and

```
Type variables: <none>

Context : N.f

pre = N.y %% 2 = 0

(1)  N.y <- N.y - 2
(2)  b' <$ {0,1}
(3)  b <- b /\ b'

post = N.y %% 2 = 0
```

and

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f

pre = ={b}

M.x <- M.x + 2              (1)  N.y <- N.y - 2
b' <$ {0,1}                 (2)  b' <$ {0,1}
b <- b /\ b'                (3)  b <- b /\ b'

post = ={b}
```

If the current goal is

```
Type variables: <none>

pre = ={b} /\ M.x{1} %% 2 + N.y{2} %% 2 = 0

    M.f ~ N.f

post = ={res} /\ M.x{1} %% 2 + N.y{2} %% 2 = 0
```

then running

```
conseq (_ : ={b} ==> ={res})
          (_ : M.x %% 2 = 0 ==> M.x %% 2 = 0)
          (_ : N.y %% 2 = 0 ==> N.y %% 2 = 0).
```

produces the goals

```
Type variables: <none>

forall &1 &2,
  ={b} /\ M.x{1} %% 2 + N.y{2} %% 2 = 0 =>
  ={b} /\ M.x{1} %% 2 = 0 /\ N.y{2} %% 2 = 0
```

and

```
Type variables: <none>

forall &1 &2,
  ={b} /\ M.x{1} %% 2 + N.y{2} %% 2 = 0 =>
```

```
forall (result_L result_R : bool) (x_L y_R : int),
  result_L = result_R /\ x_L %% 2 = 0 /\ y_R %% 2 = 0 =>
  result_L = result_R /\ x_L %% 2 + y_R %% 2 = 0
```

and

```
Type variables: <none>

pre = M.x %% 2 = 0

    M.f

post = M.x %% 2 = 0
```

and

```
Type variables: <none>

pre = N.y %% 2 = 0

    N.f

post = N.y %% 2 = 0
```

and

```
Type variables: <none>

pre = ={b}

    M.f ~ N.f

post = ={res}
```

And given lemmas

```
lemma M_N : equiv[M.f ~ N.f : ={b} ==> ={res}].
```

and

```
lemma M : hoare[M.f : M.x %% 2 = 0 ==> M.x %% 2 = 0].
```

and

```
lemma N : hoare[N.f : N.y %% 2 = 0 ==> N.y %% 2 = 0].
```

if the current goal is

```
Type variables: <none>

pre = ={b} /\ M.x{1} %% 2 + N.y{2} %% 2 = 0

    M.f ~ N.f

post = ={res} /\ M.x{1} %% 2 + N.y{2} %% 2 = 0
```

then running

```
conseq M_N M N.
```

produces the goals

```
Type variables: <none>

------------------------------------------------------------------------
forall &1 &2,
  ={b} /\ M.x{1} %% 2 + N.y{2} %% 2 = 0 =>
  ={b} /\ M.x{1} %% 2 = 0 /\ N.y{2} %% 2 = 0
```

and

```
Type variables: <none>

------------------------------------------------------------------------
forall &1 &2,
  ={b} /\ M.x{1} %% 2 + N.y{2} %% 2 = 0 =>
  forall (result_L result_R : bool) (x_L y_R : int),
    result_L = result_R /\ x_L %% 2 = 0 /\ y_R %% 2 = 0 =>
    result_L = result_R /\ x_L %% 2 + y_R %% 2 = 0
```

case

Syntax: `case` *e*. If the goal's conclusion is a pRHL, HL or pHL *statement* judgement and *e* is well-typed in the goal's context, split the goal into two goals:

- a first goal in which *e* is added as a conjunct to the conclusion's precondition; and
- a second goal in which !*e* is added as a conjunct to the conclusion's precondition.

For example, if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : N.f

pre = ={x, y}

if (x < y) {                (1--)  if (y <= x) {
  z <- y - x                (1.1)    z <- x - y
} else {                    (1--)  } else {
  z <- x - y                (1?1)    z <- y - x
}                           (1--)  }

post = ={z}
```

then running

```
case (x{1} < y{1}).
```

produces the goals

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : N.f

pre = ={x, y} /\ x{1} < y{1}

if (x < y) {                (1--)  if (y <= x) {
```

```
  z <- y - x                 (1.1)    z <- x - y
} else {                     (1--)  } else {
  z <- x - y                 (1?1)    z <- y - x
}                            (1--)  }

post = ={z}
```

and

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f

pre = ={x, y} /\ ! x{1} < y{1}

if (x < y) {                 (1--)  if (y <= x) {
  z <- y - x                 (1.1)    z <- x - y
} else {                     (1--)  } else {
  z <- x - y                 (1?1)    z <- y - x
}                            (1--)  }

post = ={z}
```

And if the current goal is

```
Type variables: <none>

Context : M.f

pre = true

(1--)  y <- -1
(2--)  while (0 < x) {
(2.1)    y <- y + 2
(2.2)    x <- x - 1
(2--)  }

post = y <> 0
```

then running

```
case (0 < x).
```

produces the goals

```
Type variables: <none>

Context : M.f

pre = 0 < x

(1--)  y <- -1
(2--)  while (0 < x) {
(2.1)    y <- y + 2
(2.2)    x <- x - 1
(2--)  }

post = y <> 0
```

and

```
Type variables: <none>

Context : M.f

pre = ! 0 < x

(1--)  y <- -1
(2--)  while (0 < x) {
(2.1)    y <- y + 2
(2.2)    x <- x - 1
(2--)  }

post = y <> 0
```

byequiv

Syntax: `byequiv (_ :` P `==>` Q`)`. If the goal's conclusion has the form

$$\texttt{Pr}[M_1.p_1(a_{1,1},\ldots,a_{1,n_1})\ \texttt{@}\ \&m_1\ :\ E_1]\ =\ \texttt{Pr}[M_2.p_2(a_{2,1},\ldots,a_{2,n_2})\ \texttt{@}\ \&m_2\ :\ E_2],$$

reduce the goal to three subgoals:

- One with conclusion `equiv`$[M_1.p_1$ ~ $M_2.p_2$: P `==>` $Q]$;
- One whose conclusion says that P holds, where references to memories `&1` and `&2` have been replaced by $\&m_1$ and $\&m_2$, respectively, and references to the formal parameters of $M_1.p_1$ and $M_2.p_2$ have been replaced by their arguments;
- One whose conclusion says that Q implies that E_1`{1}` `<=>` E_2`{2}`.

The argument to `byequiv` may be replaced by a proof term for `equiv`$[M_1.p_1$ ~ $M_2.p_2$: P `==>` $Q]$, in which case the first subgoal isn't generated. Furthermore, either or both of P and Q may be replaced by `_`, asking that the pre- or postcondition be inferred. Supplying no argument to `byequiv` is the same as replacing both P and Q by `_`. By default, inference of Q attempts to infer a conjuction of equalities implying E_1`{1}` `<=>` E_2`{2}`. Passing the `[-eq]` option to `byequiv` takes Q to *be* E_1`{1}` `<=>` E_2`{2}`.

The other variants of the tactic behave similarly with regards to the use of proof terms and specification inference.

For example, consider the module

```
module M = {
  var x : int
  proc f(y : int) : int = {
    var z : int;
    z <$ [x .. y];
    return z;
  }
}.
```

If the current goal is

```
Type variables: <none>

y1: int
y2: int
&m1: memory
&m2: memory
```

```
x_eq: M.x{m1} = M.x{m2}
y_eq: y1 = y2
```

```
Pr[M.f(y1) @ &m1 : res = 0] = Pr[M.f(y2) @ &m2 : res = 0]
```

then running

```
byequiv (_ : ={M.x, y} ==> ={res}).
```

produces the goals

```
Type variables: <none>

y1: int
y2: int
&m1: memory
&m2: memory
x_eq: M.x{m1} = M.x{m2}
y_eq: y1 = y2
```

```
pre = ={M.x, y}

    M.f ~ M.f

post = ={res}
```

and

```
Type variables: <none>

y1: int
y2: int
&m1: memory
&m2: memory
x_eq: M.x{m1} = M.x{m2}
y_eq: y1 = y2
```

```
M.x{m1} = M.x{m2} /\ y1 = y2
```

and

```
Type variables: <none>

y1: int
y2: int
&m1: memory
&m2: memory
x_eq: M.x{m1} = M.x{m2}
y_eq: y1 = y2
```

```
forall &1 &2, ={res} => res{1} = 0 <=> res{2} = 0
```

Given the lemma

```
lemma M_M_f : equiv[M.f ~ M.f : ={M.x, y} ==> ={res}].
```

if the current goal is

```
Type variables: <none>

y1: int
y2: int
&m1: memory
```

```
&m2: memory
x_eq: M.x{m1} = M.x{m2}
y_eq: y1 = y2
-----------------------------------------------------------
Pr[M.f(y1) @ &m1 : res = 0] = Pr[M.f(y2) @ &m2 : res = 0]
```

then running

```
byequiv M_M_f.
```

produces the goals

```
Type variables: <none>

y1: int
y2: int
&m1: memory
&m2: memory
x_eq: M.x{m1} = M.x{m2}
y_eq: y1 = y2
-----------------------------------------------------------
M.x{m1} = M.x{m2} /\ y1 = y2
```

and

```
Type variables: <none>

y1: int
y2: int
&m1: memory
&m2: memory
x_eq: M.x{m1} = M.x{m2}
y_eq: y1 = y2
-----------------------------------------------------------
forall &1 &2, ={res} => res{1} = 0 <=> res{2} = 0
```

And, if the current goal is

```
Type variables: <none>

y1: int
y2: int
&m1: memory
&m2: memory
x_eq: M.x{m1} = M.x{m2}
y_eq: y1 = y2
-----------------------------------------------------------
Pr[M.f(y1) @ &m1 : res = 0] = Pr[M.f(y2) @ &m2 : res = 0]
```

then running

```
byequiv.
```

produces the goals

```
Type variables: <none>

y1: int
y2: int
&m1: memory
&m2: memory
x_eq: M.x{m1} = M.x{m2}
y_eq: y1 = y2
```

```
pre = y{2} = y2 /\ M.x{2} = M.x{m2} /\ y{1} = y1 /\ M.x{1} = M.x{m1}

    M.f ~ M.f

post = ={res}
```

and

```
Type variables: <none>

y1: int
y2: int
&m1: memory
&m2: memory
x_eq: M.x{m1} = M.x{m2}
y_eq: y1 = y2
------------------------------------------------------------------------
y2 = y2 /\ M.x{m2} = M.x{m2} /\ y1 = y1 /\ M.x{m1} = M.x{m1}
```

and

```
Type variables: <none>

y1: int
y2: int
&m1: memory
&m2: memory
x_eq: M.x{m1} = M.x{m2}
y_eq: y1 = y2
------------------------------------------------------------------------
forall &1 &2, ={res} => res{1} = 0 <=> res{2} = 0
```

Syntax: `byequiv (_ :` P `==>` Q`)`. If the goal's conclusion has the form

`Pr[`M_1`.`p_1`(`$a_{1,1}$`,...,`a_{1,n_1}`) @ &`m_1 `:` E_1`] <= Pr[`M_2`.`p_2`(`$a_{2,1}$`,...,`a_{2,n_2}`) @ &`m_2 `:` E_2`]`,

then `byequiv` behaves the same as in the first variant except that the conclusion of the third subgoal says that Q implies E_1`{1} =>` E_2`{2}`.

For example, if the current goal is

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
Pr[M.f() @ &m : res] <= Pr[N.f() @ &m : res]
```

then running

```
byequiv (_ : true ==> res{1} => res{2}).
```

produces the goals

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
pre = true

    M.f ~ N.f

post = res{1} => res{2}
```

and

```
Type variables: <none>

&m: memory
------------------------------------------------
true
```

and

```
Type variables: <none>

&m: memory
------------------------------------------------
forall &1 &2, (res{1} => res{2}) => res{1} => res{2}
```

And, if the current goal is

```
Type variables: <none>

&m: memory
------------------------------------------------
Pr[M.f() @ &m : res] <= Pr[N.f() @ &m : res]
```

then running

```
byequiv.
```

produces the goals

```
Type variables: <none>

&m: memory
------------------------------------------------
pre = true

    M.f ~ N.f

post = res{1} => res{2}
```

and

```
Type variables: <none>

&m: memory
------------------------------------------------
true
```

and

```
Type variables: <none>

&m: memory
------------------------------------------------
forall &1 &2, (res{1} => res{2}) => res{1} => res{2}
```

Syntax: `byequiv` (_ : P ==> Q). If the goal's conclusion has the form

$$\texttt{Pr}[M_1.p_1(a_{1,1},\ldots,a_{1,n_1}) \texttt{ @ } \&m_1 : E_1] \texttt{ <= }$$
$$\texttt{Pr}[M_2.p_2(a_{2,1},\ldots,a_{2,n_2}) \texttt{ @ } \&m_2 : E_2] \texttt{ + } \texttt{Pr}[M_2.p_2(a_{2,1},\ldots,a_{2,n_2}) \texttt{ @ } \&m_2 : B_2],$$

then `byequiv` behaves the same as in the first variant except that the conclusion of the third subgoal says that Q implies $!B_2\{2\}$ => $E_1\{1\}$ => $E_2\{2\}$.

For example, if the current goal is

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
Pr[M.f() @ &m : res] <= Pr[N.f() @ &m : res] + Pr[N.f() @ &m : N.bad]
```

then running

```
byequiv (_ : true ==> ! N.bad{2} => res{1} => res{2}).
```

produces the goals

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
pre = true

    M.f ~ N.f

post = !N.bad{2} => res{1} => res{2}
```

and

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
true
```

FiXme Fatal: Why is the second subgoal pruned? (Compare with first and second variants, where the corresponding subgoal isn't pruned.) And, if the current goal is

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
Pr[M.f() @ &m : res] <= Pr[N.f() @ &m : res] + Pr[N.f() @ &m : N.bad]
```

then running

```
byequiv.
```

produces the goals

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
pre = true

    M.f ~ N.f

post = !N.bad{2} => res{1} => res{2}
```

and

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
true
```

FiXme Fatal: Why is the second subgoal pruned?

Syntax: `byequiv (_ : P ==> Q) : B1`. If the goal's conclusion has the form

$$\texttt{`|}\ \texttt{Pr}[M_1.p_1(a_{1,1},\ldots,a_{1,n_1})\ \texttt{@}\ \&m_1\ :\ E_1]\ \texttt{-}\ \texttt{Pr}[M_2.p_2(a_{2,1},\ldots,a_{2,n_2})\ \texttt{@}\ \&m_2\ :\ E_2]\ \texttt{|}\ \texttt{<=}\ \texttt{Pr}[M_2.p_2(a_{2,1},\ldots,a_{2,n_2})\ \texttt{@}\ \&m_2\ :\ B_2],$$

then `byequiv` behaves the same as in the first variant except that the conclusion of the third subgoal says that Q implies

$$(B_1\{1\}\ \texttt{<=>}\ B_2\{2\})\ \texttt{/\textbackslash}\ \texttt{!}\ B_2\{2\}\ \texttt{=>}\ (\ E_1\{1\}\ \texttt{<=>}\ E_2\{2\})$$

For example, if the current goal is

```
Type variables: <none>

&m: memory

`|Pr[M.f() @ &m : res] - Pr[N.f() @ &m : res]| <= Pr[N.f() @ &m : N.bad]
```

then running

```
byequiv (_ : true ==> M.bad{1} = N.bad{2} /\ (! N.bad{2} => ={res})) : M.bad
```

produces the goals

```
Type variables: <none>

&m: memory

pre = true

    M.f ~ N.f

post = M.bad{1} = N.bad{2} /\ (!N.bad{2} => ={res})
```

and

```
Type variables: <none>

&m: memory

true
```

and

```
Type variables: <none>

&m: memory

forall &1 &2,
  M.bad{1} = N.bad{2} /\ (!N.bad{2} => ={res}) =>
  (M.bad{1} <=> N.bad{2}) /\ (!N.bad{2} => res{1} <=> res{2})
```

Given the lemma

```
lemma L2 &m :
```

if the current goal is

```
Type variables: <none>

&m: memory

`|Pr[M.f() @ &m : res] - Pr[N.f() @ &m : res]| <= Pr[N.f() @ &m : N.bad]
```

then running

```
byequiv M_N_f : M.bad.
```

produces the goals

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
true
```

and

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
forall &1 &2,
  M.bad{1} = N.bad{2} /\ (!N.bad{2} => ={res}) =>
  (M.bad{1} <=> N.bad{2}) /\ (!N.bad{2} => res{1} <=> res{2})
```

byhoare

Syntax: `byhoare (_ :` P `==>` Q`)`. If the goal's conclusion has the form

$$\texttt{Pr}[M.p(a_1,\ldots,a_n)\ \texttt{@}\ \&m\ :\ E]\ \texttt{=}\ e,$$

reduce the goal to three subgoals:

- One with conclusion `phoare`$[M.p\ :\ P$ `==>` $Q]$ `=` e;
- One whose conclusion says that P holds, where variables references are looked-up in $\&m$ and references to the formal parameters of $M.p$ have been replaced by its arguments; and
- One whose conclusion says that Q `<=>` E.

The argument to `byhoare` may be replaced by a proof term for `phoare`$[M.p\ :\ P$ `==>` $Q]$ `=` e, in which case the first subgoal isn't generated. Furthermore, either or both of P and Q may be replaced by `_`, asking that the pre- or postcondition be inferred. Supplying no argument to `byhoare` is the same as replacing both P and Q by `_`.

The other variants of the tactic behave similarly with regards to the use of proof terms and specification inference.

For example, consider the module

```
module M = {
  var x : int
  proc f(y : int) : int = {
    var z : int;
    x <$ [x .. y];
    return x;
  }
```

If the current goal is

```
Type variables: <none>

x': int
y': int
&m: memory
range: x' + 1 = y'
x_eq: M.x{m} = x'
------------------------------------------------------------------------
Pr[M.f(y') @ &m : res = x'] = 1%r / 2%r
```

then running

```
byphoare (_ : M.x + 1 = y /\ x' = M.x ==> x' = res).
```

produces the goals

```
Type variables: <none>

x': int
y': int
&m: memory
range: x' + 1 = y'
x_eq: M.x{m} = x'
------------------------------------------------------------------------
pre = M.x + 1 = y /\ x' = M.x

    M.f
    [=] 1%r / 2%r

post = x' = res
```

and

```
Type variables: <none>

x': int
y': int
&m: memory
range: x' + 1 = y'
x_eq: M.x{m} = x'
------------------------------------------------------------------------
M.x{m} + 1 = y' /\ x' = M.x{m}
```

and

```
Type variables: <none>

x': int
y': int
&m: memory
range: x' + 1 = y'
x_eq: M.x{m} = x'
------------------------------------------------------------------------
forall &hr, x' = res{hr} <=> res{hr} = x'
```

Given the lemma

```
lemma M_f (x' : int) :
  phoare[M.f : M.x + 1 = y /\ x' = M.x ==> x' = res] = (1%r / 2%r).
```

if the current goal is

```
Type variables: <none>

x': int
y': int
&m: memory
range: x' + 1 = y'
x_eq: M.x{m} = x'
------------------------------------------------------------------------
Pr[M.f(y') @ &m : res = x'] = 1%r / 2%r
```

then running

```
byphoare (M_f x').
```

produces the goals

```
Type variables: <none>

x': int
y': int
&m: memory
range: x' + 1 = y'
x_eq: M.x{m} = x'
------------------------------------------------------------------------
M.x{m} + 1 = y' /\ x' = M.x{m}
```

and

```
Type variables: <none>

x': int
y': int
&m: memory
range: x' + 1 = y'
x_eq: M.x{m} = x'
------------------------------------------------------------------------
forall &hr, x' = res{hr} <=> res{hr} = x'
```

Syntax: `byphoare` `(_ :` P `==>` Q`)`. If the goal's conclusion has the form

$$e \ \texttt{<=}\ \texttt{Pr}[M.p(a_1, \ldots, a_n) \ \texttt{@}\ \&m \ :\ E],$$

then `byphoare` behaves as in the first variant except the conclusion of the first subgoal is `phoare`$[M.p\ :\ P$ `==>` $Q]$ `>=` e.

For example, if the current goal is

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
1%r / 2%r <= Pr[M.f() @ &m : res]
```

then running

```
byphoare (_ : true ==> res).
```

produces the goals

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
pre = true

    M.f
    [>=] 1%r / 2%r

post = res
```

and

```
Type variables: <none>
```

```
&m: memory
------------------------------------------------------------
true
```

and

```
Type variables: <none>

&m: memory
------------------------------------------------------------
forall _, true
```

FiXme Fatal: It's confusing how the third goal has been simplified, but not pruned.
Given the lemma

```
lemma M_f : phoare[M.f : true ==> res] >= (1%r / 2%r).
```

if the current goal is

```
Type variables: <none>

&m: memory
------------------------------------------------------------
1%r / 2%r <= Pr[M.f() @ &m : res]
```

then running

```
byphoare M_f.
```

produces the goals

```
Type variables: <none>

&m: memory
------------------------------------------------------------
true
```

and

```
Type variables: <none>

&m: memory
------------------------------------------------------------
forall _, true
```

And, if the current goal is

```
Type variables: <none>

&m: memory
------------------------------------------------------------
1%r / 2%r <= Pr[M.f() @ &m : res]
```

then running

```
byphoare.
```

produces the goals

```
Type variables: <none>

&m: memory
------------------------------------------------------------
pre = true
```

```
    M.f
    [>=] 1%r / 2%r

post = res
```

and

```
Type variables: <none>

&m: memory

true
```

and

```
Type variables: <none>

&m: memory

forall _, true
```

Syntax: `byphoare (_ :` P `==>` Q`)`. If the goal's conclusion has the form

$$\texttt{Pr}[M.p(a_1,\ldots,a_n)\ \texttt{@}\ \&m\ :\ E] \texttt{ <= } e,$$

then `byphoare` behaves as in the first variant except the conclusion of the first subgoal is `phoare`[$M.p$: P `==>` Q] `<=` e.

For example, if the current goal is

```
Type variables: <none>

&m: memory

Pr[M.f() @ &m : res] <= 1%r / 2%r
```

then running

```
byphoare (_ : true ==> res).
```

produces the goals

```
Type variables: <none>

&m: memory

pre = true

    M.f
    [<=] 1%r / 2%r

post = res
```

and

```
Type variables: <none>

&m: memory

true
```

and

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
forall _, true
```

FiXme Fatal: It's confusing how the third goal has been simplified, but not pruned.
Given the lemma

```
lemma M_f : phoare[M.f : true ==> res] <= (1%r / 2%r).
```

if the current goal is

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
Pr[M.f() @ &m : res] <= 1%r / 2%r
```

then running

```
byphoare M_f.
```

produces the goals

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
true
```

and

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
forall _, true
```

And, if the current goal is

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
Pr[M.f() @ &m : res] <= 1%r / 2%r
```

then running

```
byphoare.
```

produces the goals

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
pre = true

    M.f
    [<=] 1%r / 2%r

post = res
```

and

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
true
```

and

```
Type variables: <none>

&m: memory
------------------------------------------------------------------------
forall _, true
```

bypr

Syntax: `bypr` e_1 e_2. If the goal's conclusion has the form

$$\texttt{equiv}[M.p \sim N.q \;:\; P \;\texttt{==>}\; Q],$$

and the e_i are expressions of the same type possibly involving memories `&1` and `&2` for $M.p$ and $N.q$, respectively, then reduce the goal to two subgoals:

- One whose conclusion says that for all memories `&1` and `&2` for $M.p$ and $N.q$, if $e_1 = e_2$, then Q holds; and
- One whose conclusion says that, for all memories `&1` and `&2` for $M.p$ and $N.q$ and values a of the common type of the e_i, if P holds, then the probability of running $M.p$ in memory `&1` and with arguments consisting of the values of its formal parameters in `&1` and terminating in a memory in which the value of e_1 (replacing references to `&1` with reference to this memory) is a is the same as the probability of running $N.q$ in memory `&2` and with arguments consisting of the values of its formal parameters in `&2` and terminating in a memory in which the value of e_2 (replacing references to `&2` with reference to this memory) is a.

For example, consider the modules

```
module M = {
  var x : bool
  proc f(y : bool) : bool = {
    var b : bool;
    b <$ {0,1};
    return b /\ y;
  }
}.

module N = {
  var x : bool
  proc f(y : bool) : bool = {
    var b, b' : bool;
    b <$ {0,1};
    b' <$ {0,1};
    return (b ^^ b') /\ y;
  }
}.
```

If the current goal is

```
Type variables: <none>
```

```
pre = ={y} /\ M.x{1} = N.x{2}

    M.f ~ N.f

post = (res{1} /\ M.x{1}) = (res{2} /\ N.x{2})
```

then running

```
bypr (res, M.x){1} (res, N.x){2}.
```

produces the goals

```
Type variables: <none>

forall &1 &2 (a : bool * bool),
  (res{1}, M.x{1}) = a =>
  (res{2}, N.x{2}) = a => (res{1} /\ M.x{1}) = (res{2} /\ N.x{2})
```

and

```
Type variables: <none>

forall &1 &2 (a : bool * bool),
  ={y} /\ M.x{1} = N.x{2} =>
  Pr[M.f(y{1}) @ &1 : (res, M.x) = a] =
  Pr[N.f(y{2}) @ &2 : (res, N.x) = a]
```

Syntax: `bypr`. If the goal's conclusion has the form

$$\texttt{hoare}[M.p\ :\ P\ \texttt{==>}\ Q],$$

then reduce the goal to one whose conclusion says that, for all memories `&m` for $M.p$ such that $P\{m\}$ holds, the probability of running $M.p$ in memory `&m` and with arguments consisting of the values of its formal parameters in `&m` and terminating in a memory satisfying `!`Q is 0.

For example, consider the module

```
module M = {
  var x : bool
  proc f(y : bool) : bool = {
    var z : bool;
    z <$ {0,1};
    return x /\ y /\ z;
  }
}.
```

If the current goal is

```
Type variables: <none>

pre = !y \/ !M.x

    M.f

post = !res
```

then running

```
bypr.
```

produces the goal

```
Type variables: <none>

forall &m, !y{m} \/ !M.x{m} => Pr[M.f(y{m}) @ &m : ! !res] = 0%r
```

exists*

Syntax: `exists*` e_1, ..., e_n. If the goal's conclusion is a pRHL, HL or pHL judgment or statement judgements and the e_i are well-typed expressions typically involving program variables (in the pRHL case, the expressions will refer to memories `&1` and `&2`), then change the conclusion's precondition P to

$$\texttt{exists } (x_1 \ldots x_n),\ x_1 = e_2 \ /\backslash \ldots /\backslash\ x_n = e_n \ /\backslash\ P.$$

The tactic can be used in conjunction with `elim*` (p. 166) when handling a procedure call using a lemma that refers to initial values of program variables. See `elim*` (p. 166) for an example of this.

For example, if the current goal is

```
Type variables: <none>

pre = ={y} /\ M.x{1} = N.x{2}

    M.f ~ N.f

post = ={res}
```

then running

```
exists* (y + 1){1}, (N.x{2} - y{1}).
```

produces the goal

```
Type variables: <none>

pre =
  exists (f f0 : int),
    (f = y{1} + 1 /\ f0 = N.x{2} - y{1}) /\
    ={y} /\ M.x{1} = N.x{2}

    M.f ~ N.f

post = ={res}
```

If the current goal is

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f

pre = ={y} /\ M.x{1} = N.x{2}

z <- (-M.x) + y              (1)  z <- N.x - y

post = M.x{1} + z{1} + M.x{1} = y{2} + z{2} + y{2}
```

then running

```
exists* (y + 1){1}, (N.x{2} - y{1}).
```

produces the goal

```
Type variables: <none>

----------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : N.f

pre =
  exists (f f0 : int),
    (f = y{1} + 1 /\ f0 = N.x{2} - y{1}) /\
    ={y} /\ M.x{1} = N.x{2}

z <- (-M.x) + y            (1)  z <- N.x - y

post = M.x{1} + z{1} + M.x{1} = y{2} + z{2} + y{2}
```

If the current goal is

```
Type variables: <none>

----------------------------------------------------------------------
pre = 0 <= y /\ 0 <= M.x

    M.f

post = 0 <= res
```

then running

```
exists* (y + M.x - 2).
```

produces the goal

```
Type variables: <none>

----------------------------------------------------------------------
pre = exists (f : int), f = y + M.x - 2 /\ 0 <= y /\ 0 <= M.x

    M.f

post = 0 <= res
```

If the current goal is

```
Type variables: <none>

----------------------------------------------------------------------
Context : M.f

pre = 0 <= y /\ 0 <= M.x

(1)  z <- (-M.x) + y

post = 0 <= M.x + z + M.x
```

then running

```
exists* (y + M.x - 2).
```

produces the goal

```
Type variables: <none>

────────────────────────────────────────────────────────────────────
Context : M.f

pre = exists (f : int), f = y + M.x - 2 /\ 0 <= y /\ 0 <= M.x

(1)  z <- (-M.x) + y

post = 0 <= M.x + z + M.x
```

elim*

If the goal's conclusion is a pRHL, HL or pHL judgement or statement judgement whose precondition has the form

$$\texttt{exists}\ (x_1 \ldots x_n),\ P,$$

then remove the existential quantification from the precondition, and universally quantify the judgement or statement judgement by the x_i.
Such existential quantifications may be introduced by `sp` (p. 81) or `exists*` (p. 164).

For example, if the current goal is

```
Type variables: <none>

────────────────────────────────────────────────────────────────────
pre =
  exists (f f0 : int),
    (f = y{1} + 1 /\ f0 = N.x{2} - y{1}) /\
    ={y} /\ M.x{1} = N.x{2}

    M.f ~ N.f

post = ={res}
```

then running

```
elim*.
```

produces the goal

```
Type variables: <none>

────────────────────────────────────────────────────────────────────
forall (f f0 : int),
  equiv[ M.f ~ N.f :
        (f = y{1} + 1 /\ f0 = N.x{2} - y{1}) /\
        ={y} /\ M.x{1} = N.x{2} ==> ={res}]
```

If the current goal is

```
Type variables: <none>

────────────────────────────────────────────────────────────────────
&1 (left ) : M.f
&2 (right) : N.f

pre =
  exists (f f0 : int),
    (f = y{1} + 1 /\ f0 = N.x{2} - y{1}) /\
```

```
    ={y} /\ M.x{1} = N.x{2}

z <- (-M.x) + y            (1)  z <- N.x - y

post = M.x{1} + z{1} + M.x{1} = y{2} + z{2} + y{2}
```

then running

```
elim*.
```

produces the goal

```
Type variables: <none>

------------------------------------------------------------------------
forall (f f0 : int),
  equiv[ M.f.z <- (-M.x{hr}) + M.f.y{hr} ~ N.f.z <-
           N.x{hr} - N.f.y{hr} :
          (f = y{1} + 1 /\ f0 = N.x{2} - y{1}) /\
          ={y} /\ M.x{1} = N.x{2} ==>
          M.x{1} + z{1} + M.x{1} = y{2} + z{2} + y{2}]
```

If the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
pre = exists (f : int), f = y + M.x - 2 /\ 0 <= y /\ 0 <= M.x

    M.f

post = 0 <= res
```

then running

```
elim*.
```

produces the goal

```
Type variables: <none>

------------------------------------------------------------------------
forall (f : int),
  hoare[ M.f : f = y + M.x - 2 /\ 0 <= y /\ 0 <= M.x ==> 0 <= res]
```

If the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
Context : M.f

pre = exists (f : int), f = y + M.x - 2 /\ 0 <= y /\ 0 <= M.x

(1)  z <- (-M.x) + y

post = 0 <= M.x + z + M.x
```

then running

```
elim*.
```

produces the goal

```
Type variables: <none>

------------------------------------------------------------------------
forall (f : int),
  hoare[ z <- (-M.x) + y :
          f = y + M.x - 2 /\ 0 <= y /\ 0 <= M.x ==> 0 <= M.x + z + M.x]
```

As a more realistic example, consider the module

```
module M = {
  proc f(x : int) : int = {
    return x + 1;
  }
  proc g(x : int) : int = {
    var y : int;
    y <@ f(x);
    return y;
  }
}.
```

and lemma

```
lemma M_f (x' : int) : hoare[M.f : x' = x ==> res = x' + 1].
```

If the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
Context : M.g

pre = x = -1

(1)  y <@ M.f(x)

post = y = 0
```

then running

```
exists* x.
```

produces the goal

```
Type variables: <none>

------------------------------------------------------------------------
Context : M.g

pre = exists (x0 : int), x0 = x /\ x = -1

(1)  y <@ M.f(x)

post = y = 0
```

from which running

```
elim*=> x'.
```

produces the goal

```
Type variables: <none>
```

```
x': int
------------------------------------------------------------------------
Context : M.g

pre = x' = x /\ x = -1

(1)  y <@ M.f(x)

post = y = 0
```

from which running

```
call (M_f x').
```

produces the goal

```
Type variables: <none>

x': int
------------------------------------------------------------------------
Context : M.g

pre = x' = x /\ x = -1


post =
  x' = x && forall (result : int), result = x' + 1 => result = 0
```

hoare

FiXme Fatal: Update waiting for overhaul of pHL.
Syntax: `hoare <spec>`. Derives a null probability from a HL judgement on the procedure involved. `hoare` can also be used to derive pHL judgments and certain probability inequalities by automatically applying `conseq` (p. 136).

Examples:

`hoare`

$$\frac{\{\mathsf{true}\}\ f\ \{\neg Q\}}{\mathbf{Pr}[m, f(\vec{a}) : Q] = 0}$$

`hoare`

$$\frac{\{\mathsf{P}\}\ f\ \{\neg Q\}}{\{P\}\ f\ \{Q\} \leq 0}$$

phoare split

FiXme Fatal: Update waiting for overhaul of pHL.
Syntax: `phoare split` $\delta_A\ \delta_B\ \delta_{AB}$. Splits a pHL judgment whose postcondition is a conjunction or disjunction into three pHL judgments following the definition of the probability of a disjunction of events.

Examples:

`phoare split` $\delta_A\ \delta_B\ \delta_{AB}$ [pHL]

$$\frac{\delta_A + \delta_B - \delta_{AB} \diamond \delta \qquad \{P\}\ c\ \{A\} \diamond \delta_A \qquad \{P\}\ c\ \{B\} \diamond \delta_B \qquad \{P\}\ c\ \{A \wedge B\} \diamond^{-1} \delta_{AB}}{\{P\}\ c\ \{A \vee B\} \diamond \delta}$$

`phoare split` $\delta_A\ \delta_B\ \delta_{AB}$ [pHL]

$$\frac{\delta_A + \delta_B - \delta_{AB} \diamond \delta \qquad \{P\}\ c\ \{A\} \diamond \delta_A \qquad \{P\}\ c\ \{B\} \diamond \delta_B \qquad \{P\}\ c\ \{A \vee B\} \diamond^{-1} \delta_{AB}}{\{P\}\ c\ \{A \wedge B\} \diamond \delta}$$

Syntax: `phoare split ! δ⊤ δ!`. Splits a pHL judgment into two judgments whose postcondition are true and the negation of the original postcondition, respectively.

Examples:

`phoare split ! δ⊤ δ!` [pHL]

$$\frac{\delta_\top - \delta_! \diamond \delta \qquad \{P\}\ c\ \{\mathsf{true}\} \diamond \delta_\top \qquad \{P\}\ c\ \{!Q\} \diamond^{-1} \delta_!}{\{P\}\ c\ \{Q\} \diamond \delta}$$

Syntax: `phoare split δA δ!A: A`. Splits a pHL judgment following an event A.

Examples:

`phoare split δA δ!A: A` [pHL]

$$\frac{\delta_A + \delta_{!A} \diamond \delta \qquad \{P\}\ c\ \{Q \wedge A\} \diamond \delta_A \qquad \{P\}\ c\ \{Q \wedge \neg A\} \diamond \delta_{!A}}{\{P\}\ c\ \{Q\} \diamond \delta}$$

3.4.4 Automated Tactics

`exfalso`

Syntax: `exfalso`. Combines `conseq` (p. 136), `byequiv` (p. 149), `byphoare` (p. 156), `hoare` (p. 169) and `bypr` (p. 162) to strengthen the precondition into false and discharge the resulting trivial goal.
For example, if the current goal is

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f

pre = true

i <- 0                    (1--)  i <- 0
j <- 0                    (2--)  j <- 0
while (i <= 0) {          (3--)  while (i <= 0) {
  i <- i - 1              (3.1)    i <- i - 1
}                         (3--)  }
while (j < 10) {          (4--)  while (j < 20) {
  i <- i + 2              (4.1)    i <- i + 3
  j <- j + 1              (4.2)    j <- j + 1
}                         (4--)  }

post = ={i}
```

then running

```
seq 3 3 : (={i} /\ 0 < i{1} /\ i{1} <= 0).
```

produces the goals

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f
```

```
pre = true

i <- 0                       (1--)  i <- 0
j <- 0                       (2--)  j <- 0
while (i <= 0) {             (3--)  while (i <= 0) {
  i <- i - 1                 (3.1)    i <- i - 1
}                            (3--)  }

post = ={i} /\ 0 < i{1} /\ i{1} <= 0
```

and

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f

pre = ={i} /\ 0 < i{1} /\ i{1} <= 0

while (j < 10) {             (1--)  while (j < 20) {
  i <- i + 2                 (1.1)    i <- i + 3
  j <- j + 1                 (1.2)    j <- j + 1
}                            (1--)  }

post = ={i}
```

The first of these goals is solved by running

```
while (={i} /\ i{1} <= 0); auto; smt.
```

And running

```
exfalso.
```

reduces the second of these goals to

```
Type variables: <none>

forall &1 &2, ={i} /\ 0 < i{1} /\ i{1} <= 0 => false
```

which `smt` solves.

FiXme Fatal: Perhaps need other examples?

`auto`

If the current goal is a pRHL, HL or pHL statement judgement, uses various program logic tactics in an attempt to reduce the goal to a simpler one. Never fails, but may fail to make any progress.
FiXme Fatal: Need better description of when the tactic is applicable and how the tactic works!

For example, if the current goal is

```
Type variables: <none>

Context : M.f

pre = 1 <= y < x
```

```
(1--)  if (y < x) {
(1.1)    z1 <- x
(1.2)    x <- y
(1.3)    y <- z1
(1--)  }
(2--)  z1 <$ [x..y]
(3--)  z2 <$ [x - 1..y + 1]

post = 0 <= z1 + z2
```

then running

```
auto.
```

produces the goal

```
Type variables: <none>

------------------------------------------------------------------------
forall &hr,
  1 <= y{hr} < x{hr} =>
  if y{hr} < x{hr} then
    forall (z1 : int),
      z1 \in [y{hr}..x{hr}] =>
      forall (z2 : int),
        z2 \in [y{hr} - 1..x{hr} + 1] => 0 <= z1 + z2
  else
    forall (z1 : int),
      z1 \in [x{hr}..y{hr}] =>
      forall (z2 : int),
        z2 \in [x{hr} - 1..y{hr} + 1] => 0 <= z1 + z2
```

which `progress;smt` is able to solve. If the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
Context : N.f

pre = 1 <= y < x

(1)  z1 <$ [x..y]
(2)  x <- x - 1
(3)  y <- y + 1
(4)  z2 <$ [x..y]
(5)  x <- x - 1
(6)  y <- y + 1

post = 0 <= z1 + z2
```

then running

```
auto.
```

produce a single goal, which `smt` is able to solve. And, if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : N.f
```

```
pre = ={x, y} /\ 1 < x{1} <= y{2}

if (y < x) {                (1--)  z1 <$ [x..y]
  z1 <- x                   (1.1)
  x <- y                    (1.2)
  y <- z1                   (1.3)
}                           (1--)
z1 <$ [x..y]                (2--)  x <- x - 1
z2 <$ [x - 1..y + 1]        (3--)  y <- y + 1
                            (4--)  z2 <$ [x..y]
                            (5--)  x <- x - 1
                            (6--)  y <- y + 1

post = z1{1} + z2{1} = z1{2} + z2{2}
```

then running

```
auto.
```

produce a single goal, which **smt** is able to solve.

sim

sim attempts to solve a goal whose conclusion is a pRHL judgement or statement judgement by working backwards, propagating and extending a conjunction of equalties between variables of the two programs, verifying that the conclusion's precondition implies the final conjuction of equalities. It's capable of working backwards through **if** and **while** statments and handing random assignments, but only when the programs are sufficiently similar (thus its name). Sometimes this process only partly succeeds, leaving a statement judgement whose programs are prefixes of the original programs.

Syntax: **sim**. Without any arguments, **sim** attemps to infer the conjuction of program variable equalities from the conclusion's postcondition.

For example, if the current goal is

```
Type variables: <none>

---------------------------------------------------------------------
&1 (left ) : M.f
&2 (right) : N.f

pre = x{1} = N.u{2} /\ y{1} + 1 = N.v{2}

i <- 0                      (1--)  j <- 0
y <- y + 1                  (2--)  while (j < N.v) {
                            (2.1)    N.u <- N.u + N.u
                            (2.2)    j <- j + 1
                            (2--)  }
while (i < y) {             (3--)
  x <- x + x                (3.1)
  i <- i + 1                (3.2)
}                           (3--)

post = x{1} = N.u{2}
```

then running

```
sim.
```

produces the goal

```
Type variables: <none>
```

```
&1 (left ) : M.f
&2 (right) : N.f

pre = x{1} = N.u{2} /\ y{1} + 1 = N.v{2}

i <- 0                      (1)  j <- 0
y <- y + 1                  (2)

post = x{1} = N.u{2} /\ y{1} = N.v{2} /\ i{1} = j{2}
```

which `auto` is able to solve.

Syntax: `sim` / ϕ : *eqs*. One may give the starting conjuction, *eqs*, of equalities explicitly, and may also specifiy an invariant ϕ on the global variables of the programs.

For example, if the current goal is

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f

pre = x{1} = N.u{2} /\ y{1} + 1 = N.v{2} /\ 0 <= y{1}

i <- 0                      (1--)  j <- 0
y <- y + 1                  (2--)  while (j < N.v) {
                            (2.1)    N.u <- N.u + N.u
                            (2.2)    j <- j + 1
                            (2--)  }
while (i < y) {             (3--)
  x <- x + x                (3.1)
  i <- i + 1                (3.2)
}                           (3--)

post = x{1} = N.u{2} /\ 1 < N.v{2} + 1
```

then running

```
sim / (0 < N.v{2}) : (x{1} = N.u{2} /\ y{1} = N.v{2}).
```

produces the goals

```
Type variables: <none>

forall &1 &2,
  (x{1} = N.u{2} /\ y{1} = N.v{2}) /\ 0 < N.v{2} =>
  x{1} = N.u{2} /\ 1 < N.v{2} + 1
```

and

```
Type variables: <none>

&1 (left ) : M.f
&2 (right) : N.f

pre = x{1} = N.u{2} /\ y{1} + 1 = N.v{2} /\ 0 <= y{1}

i <- 0                      (1)  j <- 0
```

```
y <- y + 1                    (2)

post =
  (x{1} = N.u{2} /\ y{1} = N.v{2} /\ i{1} = j{2}) /\ 0 < N.v{2}
```

which `smt` and `auto;smt`, respectively, are able to solve.

Syntax: `sim` *proceq*$_1$ … *proceq*$_1$ / ϕ : *eqs*. In its most general form, one may also supply a sequence of procedure global equality specifications of the form

$$(M.p \sim N.q \; : \; eqs),$$

where *eqs* is a conjuction of global variable equalities. When `sim` encounters a pair of procedure calls consisting of a call to $M.p$ in the first program and $N.q$ in the second program, it will generate a subgoal whose conclusion is a pRHL judgment between $M.p$ and $N.q$, whose precondition assumes equality of its arguments, *eqs* and ϕ, and whose postcondition requires equality of the calls' results, *eqs* and ϕ.
One may also replace $M.p \sim N.q$ by _, meaning that the same conjunction of global variable equalities is used for all procedure calls.

For example, if the current goal is

```
Type variables: <none>

---------------------------------------------------------------
&1 (left ) : M.h
&2 (right) : N.h

pre = true

k <- 0                        (1--)  k <- 0
M.x <- 0                      (2--)  N.x <- 0
M.i <- 0                      (3--)  N.i <- 0
M.j <- 0                      (4--)  N.j <- 0
while (k < 10) {              (5--)  while (k < 10) {
  M.f()                       (5.1)    N.f()
  k <- k + 1                  (5.2)    k <- k + 1
}                             (5--)  }
while (k < 20) {              (6--)  while (k < 20) {
  M.g()                       (6.1)    N.g()
  k <- k + 1                  (6.2)    k <- k + 1
}                             (6--)  }

post = `|M.x{1} + k{1}| = `|N.x{2} - k{2}|
```

then running

```
sim (M.f ~ N.f : M.i{1} = N.i{2})
       (M.g ~ N.g : M.j{1} = N.j{2}) /
       (M.x{1} = -N.x{2}) : (={k}).
```

produces the goals

```
Type variables: <none>

---------------------------------------------------------------
forall &1 &2,
  ={k} /\ M.x{1} = -N.x{2} => `|M.x{1} + k{1}| = `|N.x{2} - k{2}|
```

and

```
Type variables: <none>
```

```
------------------------------------------------------------------------
pre = true /\ M.i{1} = N.i{2} /\ M.x{1} = -N.x{2}

    M.f ~ N.f

post = ={res} /\ M.i{1} = N.i{2} /\ M.x{1} = -N.x{2}
```

and

```
Type variables: <none>

------------------------------------------------------------------------
pre = true /\ M.j{1} = N.j{2} /\ M.x{1} = -N.x{2}

    M.g ~ N.g

post = ={res} /\ M.j{1} = N.j{2} /\ M.x{1} = -N.x{2}
```

and

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.h
&2 (right) : N.h

pre = true

k <- 0                      (1)  k <- 0
M.x <- 0                    (2)  N.x <- 0

post = ={k} /\ M.x{1} = -N.x{2}
```

which `smt`, `proc;auto;smt`, `proc;auto;smt` and `auto`, respectively, are able to solve. And, if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : M.h
&2 (right) : N.h

pre = true

k <- 0                      (1--)  k <- 0
M.x <- 0                    (2--)  N.x <- 0
M.i <- 0                    (3--)  N.i <- 0
M.j <- 0                    (4--)  N.j <- 0
while (k < 10) {            (5--)  while (k < 10) {
  M.f()                     (5.1)    N.f()
  k <- k + 1                (5.2)    k <- k + 1
}                           (5--)  }
while (k < 20) {            (6--)  while (k < 20) {
  M.g()                     (6.1)    N.g()
  k <- k + 1                (6.2)    k <- k + 1
}                           (6--)  }

post = `|M.x{1} + k{1}| = `|N.x{2} - k{2}|
```

then running

```
sim (_ : M.i{1} = N.i{2} /\ M.j{1} = N.j{2}) /
       (M.x{1} = -N.x{2}) : (={k}).
```

produces the goals

```
Type variables: <none>

--------------------------------------------------------------------------
forall &1 &2,
  ={k} /\ M.x{1} = -N.x{2} => `|M.x{1} + k{1}| = `|N.x{2} - k{2}|
```

and

```
Type variables: <none>

--------------------------------------------------------------------------
pre =
  true /\ (M.j{1} = N.j{2} /\ M.i{1} = N.i{2}) /\ M.x{1} = -N.x{2}

    M.f ~ N.f

post =
  ={res} /\
  (M.j{1} = N.j{2} /\ M.i{1} = N.i{2}) /\ M.x{1} = -N.x{2}
```

and

```
Type variables: <none>

--------------------------------------------------------------------------
pre =
  true /\ (M.j{1} = N.j{2} /\ M.i{1} = N.i{2}) /\ M.x{1} = -N.x{2}

    M.g ~ N.g

post =
  ={res} /\
  (M.j{1} = N.j{2} /\ M.i{1} = N.i{2}) /\ M.x{1} = -N.x{2}
```

and

```
Type variables: <none>

--------------------------------------------------------------------------
&1 (left ) : M.h
&2 (right) : N.h

pre = true

k <- 0                      (1)  k <- 0
M.x <- 0                    (2)  N.x <- 0

post = ={k} /\ M.x{1} = -N.x{2}
```

which `smt`, `proc;auto;smt`, `proc;auto;smt` and `auto`, respectively, are able to solve.

3.4.5 Advanced Tactics

`fel`

Syntax: `fel` *init ctr stepub bound bad conds inv*. "fel" stands for "failure event lemma". To use this tactic, one must load the theory `FelTactic`. To be applicable, the current goal's conclusion must have the form

$$\texttt{Pr}[M.p(a_1, \ldots, a_r) \texttt{ @ } \&m : \phi] \texttt{ <= } ub.$$

Here:

- ub ("upper bound") is an expression of type `real`.
- ctr is the *counter*, an expression of type `int` involving program variables.
- bad is an expression of type `bool` involving program variables. It is the "bad" or "failure" event.
- inv is an optional invariant on program variables; if it's omitted, `true` is used.
- $init$ is a natural number no bigger than the number of statements in $M.p$. It is the length of the initial part of the procedure that "initializes" the failure event lemma—causing ctr to become 0 and bad to become `false` and establishing inv. The non-initialization part of the procedure may not *directly* use the program variables on which ctr, bad and inv depend. These variables may only be modified by concrete procedures $M.p$ may directly or indirectly call—such procedures are called *oracle procedures*. If $M.p$ directly or indirectly calls an abstract procedure, there must be a module constraint saying that the abstract procedure may not modify the program variables determining the values of ctr, bad and inv or that are used by the oracle procedures.
- $bound$ is an expression of type `int`. It must be the case that

$$\phi \texttt{ /\textbackslash } inv \texttt{ => } bad \texttt{ /\textbackslash } ctr \texttt{ <= } bound.$$

- $conds$ is a list of *procedure preconditions*

$$[N_1.p_1 : \phi_1; \ldots; N_l.p_l : \phi_l],$$

  where the $N_i.p_i$ are procedures, and the ϕ_i are expressions of type `bool` involving program variables and procedure parameters. When a procedure's precondition is true, it must increase the counter's value; when it isn't true, it must not decrease the counter's value, and must preserve the value of bad. Whether a procedure's precondition holds or not, the invariant inv must be preserved.

- $stepub$ is a function of type `int -> real`, providing an upper bound as a function of the counter's current value. When a procedure's precondition, the invariant inv, `!`bad and `0 <=` ctr `<` $bound$ hold, the probability that bad becomes set during that call must be upper-bounded by the application of $stepub$ to the counter's value. In addition, it must be the case that the summation of $stepub\ i$, as i ranges from 0 to $bound - 1$, is upper-bounded by ub.

The subgoals generated by `fel` enforce the above rules. The best way to understand the details is via an example.

For example, consider the declarations

```
op upp : { int | 1 <= upp } as ge1_upp.
op n : { int | 0 <= n } as ge0_n.

module type OR = {
  proc init() : unit
  proc gen() : bool
  proc add(_ : int) : unit
}.

module Or : OR = {
  var won : bool
  var gens : int list
```

```
  proc init() : unit = {
    won <- false;
    gens <- [];
  }

  proc gen() : bool = {
    var x : int;
    if (size gens < n) {
      x <$ [1 .. upp];
      if (mem gens x) {
        won <- true;
      }
      gens <- x :: gens;
    }
    return won;
  }

  proc add(x : int) : unit = {
    if (size gens < n /\ 1 <= x <= upp) {
      gens <- x :: gens;
    }
  }
}.

module type ADV (O : OR) = {
  proc * main() : unit {O.gen O.add}
}.

module G(Adv : ADV) = {
  proc main() : unit = {
    Or.init();
    Adv(Or).main();
  }
}.
```

Here, the oracle has a boolean variable `won`, which is the bad event. It also has a list of integers `gens`, all of which are within the range 1 to `upp`, inclusive—the integers "generated" so far. The counter is the size of `gens`. The procedure `gen` randomly generates such an integer, setting `won` to `true` if the integer was previously generated. And the procedure `add` adds a new integer to the list of generated integers, without possibily setting `bad`. Both `gen` and `add` do nothing when the counter reaches the bound `n`. The adversary has access to both `gen` and `bad`.
If the current goal is

```
Type variables: <none>

Adv: ADV{Or}
&m: memory
------------------------------------------------------------------------
Pr[G(Adv).main() @ &m : Or.won] <= (n * (n - 1))%r / (2 * upp)%r
```

then running

```
fel
     1
     (size Or.gens)
     (fun ctr => ctr%r * (1%r / upp%r))
     n
     Or.won
     [Or.gen : (size Or.gens < n);
      Or.add : (size Or.gens < n /\ 1 <= x <= upp)]
```

```
     (size Or.gens <= n).
```

produces the goals

```
Type variables: <none>

Adv: ADV{Or}
&m: memory
------------------------------------------------------------------------
bigi predT (fun (ctr : int) => ctr%r * (1%r / upp%r)) 0 n <=
(n * (n - 1))%r / (2 * upp)%r
```

and

```
Type variables: <none>

Adv: ADV{Or}
&m: memory
------------------------------------------------------------------------
forall &m0,
  Or.won{m0} =>
  size Or.gens{m0} <= n => Or.won{m0} /\ size Or.gens{m0} <= n
```

and

```
Type variables: <none>

Adv: ADV{Or}
&m: memory
------------------------------------------------------------------------
Context : G(Adv).main

pre = tt = tt /\ Or.gens = Or.gens{m} /\ Or.won = Or.won{m}

(1)  Or.init()

post = (!Or.won /\ size Or.gens = 0) /\ size Or.gens <= n
```

and

```
Type variables: <none>

Adv: ADV{Or}
&m: memory
------------------------------------------------------------------------
pre =
  (((0 <= size Or.gens /\ size Or.gens < n) /\ !Or.won) /\
   size Or.gens <= n) /\
  size Or.gens < n /\ 1 <= x <= upp

    Or.add
    [<=] size Or.gens%r * (1%r / upp%r)

post = Or.won
```

and

```
Type variables: <none>

Adv: ADV{Or}
&m: memory
------------------------------------------------------------------------
forall (c : int),
```

```
  hoare[ Or.add :
          ((size Or.gens < n /\ 1 <= x <= upp) /\ c = size Or.gens) /\
          size Or.gens <= n ==>
          c < size Or.gens /\ size Or.gens <= n]
```

and

```
Type variables: <none>

Adv: ADV{Or}
&m: memory

forall (b : bool) (c : int),
  hoare[ Or.add :
          (! (size Or.gens < n /\ 1 <= x <= upp) /\
           Or.won = b /\ size Or.gens = c) /\
          size Or.gens <= n ==>
          (Or.won = b /\ c <= size Or.gens) /\ size Or.gens <= n]
```

and

```
Type variables: <none>

Adv: ADV{Or}
&m: memory

pre =
  (((0 <= size Or.gens /\ size Or.gens < n) /\ !Or.won) /\
   size Or.gens <= n) /\
  size Or.gens < n

    Or.gen
    [<=] size Or.gens%r * (1%r / upp%r)

post = Or.won
```

and

```
Type variables: <none>

Adv: ADV{Or}
&m: memory

forall (c : int),
  hoare[ Or.gen :
          (size Or.gens < n /\ c = size Or.gens) /\
          size Or.gens <= n ==>
          c < size Or.gens /\ size Or.gens <= n]
```

and

```
Type variables: <none>

Adv: ADV{Or}
&m: memory

forall (b : bool) (c : int),
  hoare[ Or.gen :
          (! size Or.gens < n /\ Or.won = b /\ size Or.gens = c) /\
          size Or.gens <= n ==>
          (Or.won = b /\ c <= size Or.gens) /\ size Or.gens <= n]
```

eager

eager is a family of tactics for proving pRHL *statement* judgements of the form

$$\texttt{equiv}[s_1\ t_1\ \texttt{\~}\ t_2\ s_2\ :\ P\ \texttt{==>}\ Q],$$

where the pre- and postconditions P and Q are conjunctions of equalities between program variables, and the statement sequences s_i only read and write global variables. Here s_1 is in the "eager" position, and its replacement, s_2, is in the "lazy" position. Some of the tactics work with *eager judgements* of the form

$$\texttt{eager}[s_1,\ M.p\ \texttt{\~}\ N.q,\ s_2\ :\ P\ \texttt{==>}\ Q],$$

where, again, the s_i only involve global variables, but where P and Q may talk about the parameters and results of $M.p$ and $N.q$ in the usual way.

The context of our examples is the following EasyCrypt script involving variable incrementation oracles `Or1` and `Or2`, which only differ in that `Or2` keeps a "transcript" of its operation.

```
module type OR = {
  proc init() : unit
  proc incr_x() : unit
  proc incr_y() : unit
  proc incr_xy() : unit
  proc incr_yx() : unit
  proc loop(n : int) : bool
}.

module Or1 : OR = {
  var x, y : int
  var b : bool
  proc init() : unit = { x <- 0; y <- 0; b <- true; }
  proc incr_x() : unit = { x <- x + 1; }
  proc incr_y() : unit = { y <- y + 1; }
  proc incr_xy() : unit = { incr_x(); incr_y(); }
  proc incr_yx() : unit = { incr_y(); incr_x(); }
  proc loop(n : int) : bool = {
    while (0 < n) {
      if (b) incr_x(); else incr_y();
      b <- !b;
    }
    return !b;
  }
}.

module Or2 : OR = {
  var x, y : int
  var b : bool
  var trace : bool list
  proc init() : unit = { x <- 0; y <- 0; b <- true; trace <- []; }
  proc incr_x() : unit = { x <- x + 1; trace <- trace ++ [true]; }
  proc incr_y() : unit = { y <- y + 1; trace <- trace ++ [false]; }
  proc incr_xy() : unit = { incr_x(); incr_y(); }
  proc incr_yx() : unit = { incr_y(); incr_x(); }
  proc loop(n : int) : bool = {
    while (0 < n) {
      if (b) incr_x(); else incr_y();
      b <- !b;
    }
    return !b;
  }
```

```
}.

lemma eager_incr :
  eager[Or1.incr_x();, Or1.incr_y ~ Or2.incr_y, Or2.incr_x(); :
        ={x, y}(Or1, Or2) ==> ={x, y}(Or1, Or2)].
proof.
eager proc.
inline *; auto.
qed.

lemma eager_incr_x :
  eager[Or1.incr_x(); Or1.incr_y();, Or1.incr_x ~
        Or2.incr_x, Or2.incr_y(); Or2.incr_x(); :
        ={x, y, b}(Or1, Or2) ==> ={x, y, b}(Or1, Or2)].
proof.
proc*.
inline*; auto.
qed.

lemma eager_incr_y :
  eager[Or1.incr_x(); Or1.incr_y();, Or1.incr_y ~
        Or2.incr_y, Or2.incr_y(); Or2.incr_x(); :
        ={x, y, b}(Or1, Or2) ==> ={x, y, b}(Or1, Or2)].
proof.
proc*; inline*; auto.
qed.

lemma eager_incr_xy :
  eager[Or1.incr_x(); Or1.incr_y();, Or1.incr_xy ~
        Or2.incr_xy, Or2.incr_y(); Or2.incr_x(); :
        ={x, y, b}(Or1, Or2) ==> ={x, y, b}(Or1, Or2)].
proof.
eager proc.
eager seq 1 1 (incr : Or1.incr_x(); Or1.incr_y(); ~
                      Or2.incr_y(); Or2.incr_x(); :
                      ={x, y}(Or1, Or2) ==> ={x, y}(Or1, Or2)) :
              (={x, y, b}(Or1, Or2)).
eager call eager_incr.
auto.
eager call eager_incr_x; first auto.
eager call eager_incr_y; first auto.
sim.
qed.

lemma eager_incr_yx :
  eager[Or1.incr_x(); Or1.incr_y();, Or1.incr_yx ~
        Or2.incr_yx, Or2.incr_y(); Or2.incr_x(); :
        ={x, y, b}(Or1, Or2) ==> ={x, y, b}(Or1, Or2)].
proof.
eager proc.
eager seq 1 1 (incr : Or1.incr_x(); Or1.incr_y(); ~
                      Or2.incr_y(); Or2.incr_x(); :
                      ={x, y}(Or1, Or2) ==> ={x, y}(Or1, Or2)) :
              (={x, y, b}(Or1, Or2));
  [eager call eager_incr; first auto |
   eager call eager_incr_y; first auto |
   eager call eager_incr_x; first auto |
   sim].
qed.
```

```
lemma eager_loop :
  eager[Or1.incr_x(); Or1.incr_y();, Or1.loop ~
        Or2.loop, Or2.incr_y(); Or2.incr_x(); :
        ={n} /\ ={x, y, b}(Or1, Or2) ==>
        ={res} /\ ={x, y, b}(Or1, Or2)].
proof.
eager proc.
swap{2} 2 2; wp.
eager while (incr : Or1.incr_x(); Or1.incr_y(); ~
                    Or2.incr_y(); Or2.incr_x(); :
                    ={n} /\ ={x, y, b}(Or1, Or2) ==>
                    ={n} /\ ={x, y, b}(Or1, Or2)).
eager call eager_incr; first auto.
trivial.
swap{2} 2 2; wp.
eager if.
trivial.
move=> &m b'; inline*; auto.
eager call eager_incr_x; first auto.
eager call eager_incr_y; first auto.
sim.
qed.

module type ADV(O : OR) = {
  proc * main() : bool {O.incr_x O.incr_y O.incr_xy O.incr_yx O.loop}
}.

module G (Adv : ADV) = {
  proc main1() : bool = {
    var b : bool;
    Or1.init();
    Or1.incr_x(); Or1.incr_y();
    b <@ Adv(Or1).main();
    return b;
  }
  proc main2() : bool = {
    var b : bool;
    Or2.init();
    b <@ Adv(Or2).main();
    Or2.incr_y(); Or2.incr_x();
    return b;
  }
}.

lemma eager_adv (Adv <: ADV{Or1, Or2}) :
  eager[Or1.incr_x(); Or1.incr_y();, Adv(Or1).main ~
        Adv(Or2).main, Or2.incr_y(); Or2.incr_x(); :
        ={x, y, b}(Or1, Or2) ==> ={res} /\ ={x, y}(Or1, Or2)].
proof.
eager proc (incr : Or1.incr_x(); Or1.incr_y(); ~
                   Or2.incr_y(); Or2.incr_x(); :
                   ={x, y}(Or1, Or2) ==> ={x, y}(Or1, Or2))
           (={x, y, b}(Or1, Or2)).
eager call eager_incr; first auto.
trivial.
trivial.
apply eager_incr_x.
sim.
apply eager_incr_y.
sim.
```

```
apply eager_incr_xy.
sim.
apply eager_incr_yx.
sim.
apply eager_loop.
sim.
qed.

lemma G_Adv (Adv <: ADV{Or1, Or2}) :
  equiv[G(Adv).main1 ~ G(Adv).main2 :
        true ==> ={res} /\ ={x, y}(Or1, Or2)].
proof.
proc.
seq 1 1 : (={x, y, b}(Or1, Or2)); first inline*; auto.
eager call (eager_adv Adv); first auto.
qed.

lemma G_Adv' (Adv <: ADV{Or1, Or2}) :
  equiv[G(Adv).main1 ~ G(Adv).main2 :
        true ==> ={res} /\ ={x, y}(Or1, Or2)].
proof.
proc.
seq 1 1 : (={x, y, b}(Or1, Or2)); first inline*; auto.
eager (incr : Or1.incr_x(); Or1.incr_y(); ~
              Or2.incr_y(); Or2.incr_x(); :
              ={x, y}(Or1, Or2) ==> ={x, y}(Or1, Or2)) :
      (={x, y, b}(Or1, Or2)).
eager call eager_incr; first auto.
auto.
eager proc incr (={x, y, b}(Or1, Or2));
  [trivial | trivial | apply eager_incr_x | sim |
   apply eager_incr_y | sim | apply eager_incr_xy | sim |
   apply eager_incr_yx | sim | apply eager_loop | sim].
qed.
```

Syntax: `eager proc`. Turn a goal whose conclusion is an eager judgement into one whose conclusion is a pRHL statement judgement in which the eager judgement's procedures have been replaced by their bodies. For example, if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
eager[ Or1.incr_x();, Or1.incr_y ~ Or2.incr_y, Or2.incr_x(
        ); :
        Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
        Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]
```

then running

```
eager proc.
```

produces the goal

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : Or1.incr_y
&2 (right) : Or2.incr_y

pre = Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}
```

```
Or1.incr_x()              (1)  Or2.y <- Or2.y + 1
Or1.y <- Or1.y + 1        (2)  Or2.trace <- Or2.trace ++ [false]
                          (3)  Or2.incr_x()

post = Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}
```

Syntax: `proc*`. Turn a goal whose conclusion is an eager judgement into one whose conclusion is a pRHL statement judgement in which the eager judgement's procedures are called, as opposed to being inlined. For example, if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
eager[ Or1.incr_x();; Or1.incr_y();, Or1.incr_x ~ Or2.incr_x,
        Or2.incr_y();; Or2.incr_x(); :
        Or1.x{1} = Or2.x{2} /\
        Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} ==>
        Or1.x{1} = Or2.x{2} /\
        Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}]
```

then running

```
proc*.
```

produces the goal

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : Or1.incr_x
&2 (right) : Or2.incr_x

pre =
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}

Or1.incr_x()              (1)  r <@ Or2.incr_x()
Or1.incr_y()              (2)  Or2.incr_y()
r <@ Or1.incr_x()         (3)  Or2.incr_x()

post =
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}
```

Syntax: `eager call` p. Here p is a proof term for an eager judgement. If the goal's conclusion is a pRHL statement judgement whose programs' suffixes match the left and right sides of the eager judgment, then consume those suffixes.

For example, if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : Or1.incr_xy
&2 (right) : Or2.incr_xy

pre = Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}

Or1.incr_x()              (1)  Or2.incr_y()
Or1.incr_y()              (2)  Or2.incr_x()
```

```
post = Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}
```

then running

```
eager call eager_incr.
```

(see the statement of `eager_incr`, above) produces the goal

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : Or1.incr_xy
&2 (right) : Or2.incr_xy

pre = Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}


post = Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}
```

Syntax: `eager seq` n_1 n_2 `(`H `:` s_1 `~` s_2 `:` A `==>` B`) :` C.
Here, the goal's conclusion must be a pRHL statement judgement whose left and right programs begin and end with s_1 and s_2, respectively. A first subgoal is generated whose conclusion is the specified pRHL statement judgement, which is made available as H for the subsequent subgoals's use. The n_1 and n_2 must be natural numbers saying how many statements from the part of the first program following s_1 and from the beginning of the second program to put together with the s_i in a second pRHL statement judgement subgoal. The remaining statements are put together with the s_i in a third pRHL statement judgement subgoal. And there is a final subgoal whose conclusion is a pRHL statement judgment whose left and right sides are those remaining statements of the second program only.

For example, if the current goal is

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : Or1.incr_xy
&2 (right) : Or2.incr_xy

pre =
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}

Or1.incr_x()               (1)  Or2.incr_x()
Or1.incr_y()               (2)  Or2.incr_y()
Or1.incr_x()               (3)  Or2.incr_y()
Or1.incr_y()               (4)  Or2.incr_x()

post =
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}
```

then running

```
eager seq 1 1 (incr : Or1.incr_x(); Or1.incr_y(); ~
                        Or2.incr_y(); Or2.incr_x(); :
                        ={x, y}(Or1, Or2) ==> ={x, y}(Or1, Or2)) :
                (={x, y, b}(Or1, Or2)).
```

produces the goals

```
Type variables: <none>

&1 (left ) : Or1.incr_xy
&2 (right) : Or2.incr_xy

pre = Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}

Or1.incr_x()               (1)  Or2.incr_y()
Or1.incr_y()               (2)  Or2.incr_x()

post = Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}
```

and

```
Type variables: <none>

incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
              Or2.incr_x(); :
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]

&1 (left ) : Or1.incr_xy
&2 (right) : Or2.incr_xy

pre =
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}

Or1.incr_x()               (1)  Or2.incr_x()
Or1.incr_y()               (2)  Or2.incr_y()
Or1.incr_x()               (3)  Or2.incr_x()

post =
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}
```

and

```
Type variables: <none>

incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
              Or2.incr_x(); :
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]

&1 (left ) : Or1.incr_xy
&2 (right) : Or2.incr_xy

pre =
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}

Or1.incr_x()               (1)  Or2.incr_y()
Or1.incr_y()               (2)  Or2.incr_y()
Or1.incr_y()               (3)  Or2.incr_x()

post =
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}
```

and

```
Type variables: <none>

incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
              Or2.incr_x(); :
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]
```

```
&1 (left ) : Or2.incr_xy [programs are in sync]
&2 (right) : Or2.incr_xy

pre = ={Or2.b, Or2.y, Or2.x}

(1)  Or2.incr_y()

post = ={Or2.b, Or2.y, Or2.x}
```

Syntax: `eager if`. If the goal's conclusion is a pRHL statement judgement whose left program consists of s_1 followed by a conditional, and whose right program consists of a conditional followed by s_2, reduce the goal to two subgoals using the s_i together with the "then" and "else" parts of the conditionals, along with auxiliary subgoals verifying that—even after running s_1—the conditionals' boolean expressions are equivalent.

For example, if the current goal is

```
Type variables: <none>

incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();; Or2.incr_x(
              ); :
              ={n} /\
              Or1.x{1} = Or2.x{2} /\
              Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} ==>
              ={n} /\
              Or1.x{1} = Or2.x{2} /\
              Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}]
```

```
&1 (left ) : Or1.loop
&2 (right) : Or2.loop

pre =
  (={n} /\ Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}) /\
  0 < n{1}

Or1.incr_x()                (1--)  if (Or2.b) {
                            (1.1)    Or2.incr_x()
                            (1--)  } else {
                            (1?1)    Or2.incr_y()
                            (1--)  }
Or1.incr_y()                (2--)  Or2.incr_y()
if (Or1.b) {                (3--)  Or2.incr_x()
  Or1.incr_x()              (3.1)
} else {                    (3--)
  Or1.incr_y()              (3?1)
}                           (3--)

post =
  ={n} /\
  Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} /\ (!Or1.b{1}) = !Or2.b{2}
```

then running

```
eager if.
```

produces the goals

```
Type variables: <none>

incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();; Or2.incr_x(
              ); :
              ={n} /\
              Or1.x{1} = Or2.x{2} /\
              Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} ==>
              ={n} /\
              Or1.x{1} = Or2.x{2} /\
              Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}]
------------------------------------------------------------------------
forall &1 &2,
  (={n} /\ Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}) /\
  0 < n{1} => Or1.b{1} = Or2.b{2}
```

and

```
Type variables: <none>

incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();; Or2.incr_x(
              ); :
              ={n} /\
              Or1.x{1} = Or2.x{2} /\
              Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} ==>
              ={n} /\
              Or1.x{1} = Or2.x{2} /\
              Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}]
------------------------------------------------------------------------
forall &m2 (b1 : bool),
  hoare[ Or1.incr_x();; Or1.incr_y(); :
          ((n = n{m2} /\
            Or1.x = Or2.x{m2} /\ Or1.y = Or2.y{m2} /\ Or1.b = Or2.b{m2}) /\
           0 < n) /\
          Or1.b = b1 ==> Or1.b = b1]
```

and

```
Type variables: <none>

incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();; Or2.incr_x(
              ); :
              ={n} /\
              Or1.x{1} = Or2.x{2} /\
              Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} ==>
              ={n} /\
              Or1.x{1} = Or2.x{2} /\
              Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}]
------------------------------------------------------------------------
&1 (left ) : Or1.loop
&2 (right) : Or2.loop

pre =
  ((={n} /\ Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}) /\
   0 < n{1}) /\
  Or1.b{hr} = true

Or1.incr_x()               (1)  Or2.incr_x()
```

```
Or1.incr_y()                (2)  Or2.incr_y()
Or1.incr_x()                (3)  Or2.incr_x()

post =
  ={n} /\
  Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} /\ (!Or1.b{1}) = !Or2.b{2}
```

and

```
Type variables: <none>

incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();; Or2.incr_x(
              ); :
              ={n} /\
              Or1.x{1} = Or2.x{2} /\
              Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} ==>
              ={n} /\
              Or1.x{1} = Or2.x{2} /\
              Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}]

&1 (left ) : Or1.loop
&2 (right) : Or2.loop

pre =
  ((={n} /\ Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}) /\
   0 < n{1}) /\
  Or1.b{hr} = false

Or1.incr_x()                (1)  Or2.incr_y()
Or1.incr_y()                (2)  Or2.incr_y()
Or1.incr_y()                (3)  Or2.incr_x()

post =
  ={n} /\
  Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} /\ (!Or1.b{1}) = !Or2.b{2}
```

Syntax: `eager while` (H : s_1 ~ s_2 : A ==> B). Like `eager if`, but working with while loops instead of conditionals and featuring an explicit, named pRHL statement judgement involving the s_i, available as H to the subgoals. The subgoal involving the bodies of the while loops uses B as its invariant. There is also a subgoal whose conclusion is a pRHL statement judgement both of whose sides are the body of the second program's while loop.

For example, if the current goal is

```
Type variables: <none>

&1 (left ) : Or1.loop
&2 (right) : Or2.loop

pre =
  ={n} /\
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}

Or1.incr_x()                (1----)  while (0 < n) {
                            (1.1--)    if (Or2.b) {
                            (1.1.1)      Or2.incr_x()
                            (1.1--)    } else {
                            (1.1?1)      Or2.incr_y()
                            (1.1--)    }
```

```
                              (1.2--)    Or2.b <- !Or2.b
                              (1----)  }
Or1.incr_y()                  (2----)  Or2.incr_y()
while (0 < n) {               (3----)  Or2.incr_x()
  if (Or1.b) {                (3.1--)
    Or1.incr_x()              (3.1.1)
  } else {                    (3.1--)
    Or1.incr_y()              (3.1?1)
  }                           (3.1--)
  Or1.b <- !Or1.b             (3.2--)
}                             (3----)

post =
  (!Or1.b{1}) = !Or2.b{2} /\
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}
```

then running

```
eager while (incr : Or1.incr_x(); Or1.incr_y(); ~
                    Or2.incr_y(); Or2.incr_x(); :
                    ={n} /\ ={x, y, b}(Or1, Or2) ==>
                    ={n} /\ ={x, y, b}(Or1, Or2)).
```

produces the goals

```
Type variables: <none>

------------------------------------------------------------------------
&1 (left ) : Or1.loop
&2 (right) : Or2.loop

pre =
  ={n} /\
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}

Or1.incr_x()                  (1)  Or2.incr_y()
Or1.incr_y()                  (2)  Or2.incr_x()

post =
  ={n} /\
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}
```

and

```
Type variables: <none>

incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
              Or2.incr_x(); :
              ={n} /\
              Or1.x{1} = Or2.x{2} /\
              Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} ==>
              ={n} /\
              Or1.x{1} = Or2.x{2} /\
              Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}]

------------------------------------------------------------------------
forall &1 &2,
  ={n} /\
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} =>
```

```
  0 < n{1} = 0 < n{2}
```

and

```
Type variables: <none>

incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
              Or2.incr_x(); :
              ={n} /\
              Or1.x{1} = Or2.x{2} /\
              Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} ==>
              ={n} /\
              Or1.x{1} = Or2.x{2} /\
              Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}]
```

```
&1 (left ) : Or1.loop
&2 (right) : Or2.loop

pre =
  (={n} /\
   Or1.x{1} = Or2.x{2} /\
   Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}) /\
  0 < n{1}

Or1.incr_x()                (1--)  if (Or2.b) {
                            (1.1)    Or2.incr_x()
                            (1--)  } else {
                            (1?1)    Or2.incr_y()
                            (1--)  }
Or1.incr_y()                (2--)  Or2.b <- !Or2.b
if (Or1.b) {                (3--)  Or2.incr_y()
  Or1.incr_x()              (3.1)
} else {                    (3--)
  Or1.incr_y()              (3?1)
}                           (3--)
Or1.b <- !Or1.b             (4--)  Or2.incr_x()

post =
  ={n} /\
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}
```

and

```
Type variables: <none>

incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
              Or2.incr_x(); :
              ={n} /\
              Or1.x{1} = Or2.x{2} /\
              Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} ==>
              ={n} /\
              Or1.x{1} = Or2.x{2} /\
              Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}]
```

```
&1 (left ) : Or2.loop [programs are in sync]
&2 (right) : Or2.loop

pre = ={n, Or2.b, Or2.y, Or2.x}

(1--)  if (Or2.b) {
```

```
(1.1)     Or2.incr_x()
(1--)   } else {
(1?1)     Or2.incr_y()
(1--)   }
(2--)   Or2.b <- !Or2.b

post = ={n, Or2.b, Or2.y, Or2.x}
```

Syntax: `eager proc (`H `:` s_1 `~` s_2 `:` A `==>` B`)` C `|` `eager` H C. This is the form of `eager proc` that applies when the procedures $M.p$ and $N.q$ are abstract. The second variant is where the specified pRHL statement judgement has already been introduced by another `eager` tactic. There must be a module restriction saying that the abstract procedures can't directly interfere with the global variables on which the s_i depend. Subgoals are generated for each pair of "oracles" the abstract procedures are capable of calling, i.e., for each of the procedures that may read/write the global variables used by the s_i.

For example, if the current goal is

```
Type variables: <none>

Adv: ADV{Or1, Or2}
------------------------------------------------------------------------
eager[ Or1.incr_x();; Or1.incr_y();, Adv(Or1).main ~
        Adv(Or2).main, Or2.incr_y();; Or2.incr_x(); :
        Or1.x{1} = Or2.x{2} /\
        Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} ==>
        ={res} /\ Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]
```

then running

```
eager proc (incr : Or1.incr_x(); Or1.incr_y(); ~
                   Or2.incr_y(); Or2.incr_x(); :
                   ={x, y}(Or1, Or2) ==> ={x, y}(Or1, Or2))
           (={x, y, b}(Or1, Or2)).
```

produces the goals

```
Type variables: <none>

Adv: ADV{Or1, Or2}
------------------------------------------------------------------------
&1 (left ) : Top.
&2 (right) : Top.

pre = Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}

Or1.incr_x()               (1)  Or2.incr_y()
Or1.incr_y()               (2)  Or2.incr_x()

post = Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
              Or2.incr_x(); :
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]
------------------------------------------------------------------------
```

```
forall &1 &2,
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} =>
  true /\
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
             Or2.incr_x(); :
             Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
             Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]
```

```
forall &1 &2,
  ={res} /\
  ={glob Adv} /\
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} =>
  ={res} /\ Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
             Or2.incr_x(); :
             Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
             Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]
```

```
eager[ Or1.incr_x();; Or1.incr_y();, Or1.incr_x ~ Or2.incr_x,
       Or2.incr_y();; Or2.incr_x(); :
       true /\
       Or1.x{1} = Or2.x{2} /\
       Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} ==>
       ={res} /\
       Or1.x{1} = Or2.x{2} /\
       Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}]
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
             Or2.incr_x(); :
             Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
             Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]
```

```
pre = ={Or2.b, Or2.y, Or2.x}

    Or2.incr_x ~ Or2.incr_x

post = ={res, Or2.b, Or2.y, Or2.x}
```

and

```
Type variables: <none>
```

```
Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
             Or2.incr_x(); :
             Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
             Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]
------------------------------------------------------------------
eager[ Or1.incr_x();; Or1.incr_y();, Or1.incr_y ~ Or2.incr_y,
        Or2.incr_y();; Or2.incr_x(); :
        true /\
        Or1.x{1} = Or2.x{2} /\
        Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} ==>
        ={res} /\
        Or1.x{1} = Or2.x{2} /\
        Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}]
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
             Or2.incr_x(); :
             Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
             Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]
------------------------------------------------------------------
pre = ={Or2.b, Or2.y, Or2.x}

    Or2.incr_y ~ Or2.incr_y

post = ={res, Or2.b, Or2.y, Or2.x}
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
             Or2.incr_x(); :
             Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
             Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]
------------------------------------------------------------------
eager[ Or1.incr_x();; Or1.incr_y();, Or1.incr_xy ~ Or2.incr_xy,
        Or2.incr_y();; Or2.incr_x(); :
        true /\
        Or1.x{1} = Or2.x{2} /\
        Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} ==>
        ={res} /\
        Or1.x{1} = Or2.x{2} /\
        Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}]
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
             Or2.incr_x(); :
             Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
             Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]
------------------------------------------------------------------
pre = ={Or2.b, Or2.y, Or2.x}
```

```
    Or2.incr_xy ~ Or2.incr_xy

post = ={res, Or2.b, Or2.y, Or2.x}
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
              Or2.incr_x(); :
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]

eager[ Or1.incr_x();; Or1.incr_y();, Or1.incr_yx ~ Or2.incr_yx,
        Or2.incr_y();; Or2.incr_x(); :
        true /\
        Or1.x{1} = Or2.x{2} /\
        Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} ==>
        ={res} /\
        Or1.x{1} = Or2.x{2} /\
        Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}]
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
              Or2.incr_x(); :
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]

pre = ={Or2.b, Or2.y, Or2.x}

    Or2.incr_yx ~ Or2.incr_yx

post = ={res, Or2.b, Or2.y, Or2.x}
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
              Or2.incr_x(); :
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]

eager[ Or1.incr_x();; Or1.incr_y();, Or1.loop ~ Or2.loop,
        Or2.incr_y();; Or2.incr_x(); :
        ={n} /\
        Or1.x{1} = Or2.x{2} /\
        Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} ==>
        ={res} /\
        Or1.x{1} = Or2.x{2} /\
        Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}]
```

and

```
Type variables: <none>
```

```
Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
             Or2.incr_x(); :
             Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
             Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]
--------------------------------------------------------------------
pre = ={n, Or2.b, Or2.y, Or2.x}

    Or2.loop ~ Or2.loop

post = ={res, Or2.b, Or2.y, Or2.x}
```

Syntax: `eager` (H : s_1 ~ s_2 : A ==> B) : C. Reduces a goal whose conclusion is a pRHL statement judgement of the form

$$\texttt{equiv}[s_1\, t_1 \sim t_2\, s_2 \ : P \texttt{ ==> } Q],$$

to an eager judgement, along with a subgoal for the specified pRHL statement judgement H, plus an auxiliary goal.

For example, if the current goal is

```
Type variables: <none>

Adv: ADV{Or1, Or2}
--------------------------------------------------------------------
&1 (left ) : G(Adv).main1
&2 (right) : G(Adv).main2

pre =
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}

Or1.incr_x()              (1)  b <@ Adv(Or2).main()
Or1.incr_y()              (2)  Or2.incr_y()
b <@ Adv(Or1).main()      (3)  Or2.incr_x()

post = ={b} /\ Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}
```

then running

```
eager (incr : Or1.incr_x(); Or1.incr_y(); ~
                 Or2.incr_y(); Or2.incr_x(); :
                 ={x, y}(Or1, Or2) ==> ={x, y}(Or1, Or2)) :
         (={x, y, b}(Or1, Or2)).
```

produces the goals

```
Type variables: <none>

Adv: ADV{Or1, Or2}
--------------------------------------------------------------------
&1 (left ) : G(Adv).main1
&2 (right) : G(Adv).main2

pre = Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}

Or1.incr_x()              (1)  Or2.incr_y()
Or1.incr_y()              (2)  Or2.incr_x()

post = Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
              Or2.incr_x(); :
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]
-------------------------------------------------------------------
&1 (left ) : G(Adv).main1
&2 (right) : G(Adv).main2

pre =
  Or1.x{1} = Or2.x{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}


post =
  Or1.b{1} = Or2.b{2} /\
  Or1.y{1} = Or2.y{2} /\ Or1.x{1} = Or2.x{2}
```

and

```
Type variables: <none>

Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); ~ Or2.incr_y();;
              Or2.incr_x(); :
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
              Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]
-------------------------------------------------------------------
eager[ Or1.incr_x();; Or1.incr_y();, Adv(Or1).main ~
        Adv(Or2).main, Or2.incr_y();; Or2.incr_x(); :
        Or1.x{1} = Or2.x{2} /\
        Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} ==>
        (Or1.y{1} = Or2.y{2} /\ Or1.x{1} = Or2.x{2}) /\ ={res}]
```

Chapter 4

Structuring Specifications and Proofs

4.1 Theories

4.2 Sections

Chapter 5

EasyCrypt Library

Chapter 6

Advanced Features and Usage

Chapter 7

Examples

7.1 Hashed ElGamal

7.2 BR93

We'll work through [BR93].

Bibliography

[BDG$^+$14] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A tutorial. In Alessandro Aldini, Javier Lopez, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer International Publishing, 2014.

[BGHZ11] Gilles Barthe, Benjamin Grégoire, Sylvain Héraud, and Santiago Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *Proceedings of the 31st Annual Conference on Advances in Cryptology*, CRYPTO'11, pages 71–90. Springer-Verlag, 2011.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, CCS'93, pages 62–73, New York, NY, USA, 1993. ACM.

[BR04] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. *IACR Cryptology ePrint Archive*, 2004(331), 2004.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Proceedings of the 24th Annual International Conference on The Theory and Applications of Cryptographic Techniques*, EUROCRYPT'06, pages 409–426. Springer-Verlag, 2006.

[Sho04] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. http://eprint.iacr.org/.

Index of Tactics

admit, 58
algebra, 68
alias, 125
apply, 60
assumption, 48
auto, 171

byequiv, 149
byphoare, 156
bypr, 162

call, 93
case, 64
case-pl, 147
cfold, 126
change, 58
clear, 47
closing goals, 71
congr, 53
conseq, 136
cut, 60

done, 56

eager, 182
elim, 67
elim*, 166
exact, 61
exfalso, 170
exists, 50
exists*, 164

failure recovery, 69
fel, 177
fission, 122
fusion, 123

generalization, 46
goal selection, 70

have, 59
hoare, 169

idtac, 47
if, 86
inline, 106
introduction, 41

kill, 128

left, 48

move, 47

phoare split, 169
pose, 59
proc, 72
progress, 57

rcondf, 113
rcondt, 108
reflexivity, 48
rewrite, 62
right, 49
rnd, 83

seq, 79
sequence, 69
sequence with branching, 69
sim, 173
simplify, 56
skip, 79
smt, 57
sp, 81
split, 50
splitwhile, 119
subst, 54
swap, 103
symmetry, 130

tactic repetition, 70
transitivity, 131
trivial, 55

unroll, 117

while, 89
wp, 82

List of Corrections

Note: Add explanation of new **`replace`** tactic, which uses statement patterns. 38
Note: Need explanation of how a proof term may be used in forward reasoning. 40
Note: Can it be used in forward reasoning? 40
Note: In forward reasoning they aren't equivalent—why? 40
Note: Be a bit more detailed about what this tactic does? 56
Note: Is this the right place to define "convertible"? 57
Note: Describe **`progress`** options. 57
Note: Describe failure states of prover selection. 57
Note: Describe *dbhint* options. 58
Note: Make this a pragma? 58
Note: Missing description of **`algebra`** 68
Fatal: Why is the second subgoal pruned? (Compare with first and second variants, where the corresponding subgoal isn't pruned.) 154
Fatal: Why is the second subgoal pruned? 155
Fatal: It's confusing how the third goal has been simplified, but not pruned. 159
Fatal: It's confusing how the third goal has been simplified, but not pruned. 161
Fatal: Update waiting for overhaul of pHL. 169
Fatal: Update waiting for overhaul of pHL. 169
Fatal: Perhaps need other examples? 171
Fatal: Need better description of when the tactic is applicable and how the tactic works! 171