## 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, Printing and Searching .............. 9  
2.3 Expressions Language .................................. 10  
  2.3.1 Type Expressions .................................. 10  
  2.3.2 Type Declarations ................................. 11  
  2.3.3 Expressions and Operator Declarations .......... 13  
2.4 Module System ......................................... 17  
  2.4.1 Modules ............................................ 17  
  2.4.2 Module Types ...................................... 21  
  2.4.3 Global Variables .................................. 26  
2.5 Logics .................................................. 26  
  2.5.1 Formulas .......................................... 26  
  2.5.2 Axioms and Lemmas ............................... 32  

3 Tactics 35  
3.1 Proof Engine .......................................... 35  
3.2 Ambient logic ......................................... 38  
  3.2.1 Proof Terms ...................................... 38  
  3.2.2 Occurrence Selectors and Rewriting Directions .. 39  
  3.2.3 Introduction and Generalization .................. 40  
  3.2.4 Tactics .......................................... 47  
  3.2.5 TacticaIs .......................................... 68  
3.3 Program Logics ........................................ 71  
  3.3.1 Tactics for Reasoning about Programs ............ 72  
  3.3.2 Tactics for Transforming Programs ............... 103  
  3.3.3 Tactics for Reasoning about Specifications ...... 125  
  3.3.4 Automated Tactics ................................ 131  
  3.3.5 Advanced Tactics .................................. 132  

4 Structuring Specifications and Proofs 133  
4.1 Theories .............................................. 133  
4.2 Sections ............................................... 133  

5 EasyCrypt Library 134
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 Advanced Features and Usage</td>
<td>135</td>
</tr>
<tr>
<td>7 Examples</td>
<td>136</td>
</tr>
<tr>
<td>7.1 Hashed ElGamal</td>
<td>136</td>
</tr>
<tr>
<td>7.2 BR93</td>
<td>136</td>
</tr>
<tr>
<td>References</td>
<td>137</td>
</tr>
<tr>
<td>Index</td>
<td>138</td>
</tr>
</tbody>
</table>
Chapter 1

Getting Started

1.1 Introduction

EASYCRYPT [BDG+14, BHZ11] 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:

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

Listing 1.1

(* Load (require) the theory of the booleans and import its symbols, into the environment. The Bool theory defines, among other symbols, the exclusive-or operator (\^\^) and the uniform distribution on booleans (\{0,1\} *)

require import Bool.

(* 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.
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. 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.

---

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 modules may refer to the global variables of previously declared modules. 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: admit, algebra, alias, apply, as, assert, assumption, auto, axiom, axiomatized, beta, by, byequiv, byhoare, bypr, call, case, cfold, change, class, clear, clone, congr, conseq, const, cut, declare, delta, do, done, eager, elf, elim, else, end, equiv, exact, exfalso, exists, expect, export, fel, fieldseq, first, fission, forall, fun, fusion, generalize, glob, goal, have, hint, hoare, hypothesis, idtac, if, import, in, inline, instance, intros, iota, islossless, kill, last, left, lemma, let, local, logic, modpath, module, move, nolocals, nosmt, of, op, phoare, pose, Pr, pr_bounded, pred, print, proc, progress, proof, prover, qed, rcondf, rcondt, realize, reflexivity, require, res, return, rewrite, right, rineq, rnd, runnormal, search, section, seq, sim, simplify, skip, smt, sp, split, splitwhile, strict, subst, swap, symmetry, then, theory, timeout, Top, transitivity, trivial, try, try, type, unroll, var, while, why3, with, 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 expect, first, last, left, right or strict.

- **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);
– $\leq$, $\geq$ (non-associative);
– $\lor\land$ (right-associative);
– $\&\&$, $\lor\land$ (right-associative);
– $=\land<>$ (non-associative);
– $<,>,<=\land>=$(left-associative);
– $-\land+$ (left-associative);
– $\ast$, 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: $\backslash$, $\|$, $\_\_\_\_\_\_\_\_$ or $\_\_\_\_\_\_\_<$. (We’ll see below how they may be used in mixfix form.)

• **Record projections.** A *record 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.

• **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 uppercase letter.

### 2.2 Script Structure, Printing and Searching

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

• declare types and type constructors;
• declare operators and predicates;
• declare modules or module types;
• state axioms or lemmas;
• apply tactics;
• require (make available) theories;
• print types, operators, predicates, modules, module types, axioms and lemmas;
• search for lemmas involving operators.

To print an entity, one may say:

- \texttt{print type t}.
- \texttt{print op f}.
- \texttt{print pred p}.
- \texttt{print module Foo}.
- \texttt{print module type FOO}.
- \texttt{print axiom foo}.
- \texttt{print lemma goo}.

The entity kind may be omitted, in which case all entities with the given name are printed. \texttt{print op} and \texttt{print pred} may be used interchangeably, and may be applied to record field projections and datatype constructors, as well as to operators and predicates—all of which share the same name space. \texttt{print axiom} and \texttt{print lemma} are also interchangeable—axioms and lemmas share the same name space.

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

- \texttt{search f}.
- \texttt{search (+)}.
- \texttt{search (+) (-). (** axioms/lemmas involving both operators *)}

(Infix operators must be parenthesized.)

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

\textsc{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 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 \textsc{EasyCrypt}'s types must be inhabited — i.e. nonempty.
TABLE 2. SPECIFICATIONS

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>type constructor application</td>
<td>—</td>
</tr>
<tr>
<td>*</td>
<td>tuple constructor</td>
</tr>
<tr>
<td>-&gt;</td>
<td>function type</td>
</tr>
</tbody>
</table>

constructions with higher precedences come first

Figure 2.2: Type operators precedence and associativity

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 \( t \text{ distr} \) 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 \( \tau \Rightarrow \sigma \) denotes the type of total functions mapping elements of type \( \tau \) to elements of type \( \sigma \). Note that \( \Rightarrow \) associates to the right, so that int \( \Rightarrow \) bool \( \Rightarrow \) real and int \( \Rightarrow \) (bool \( \Rightarrow \) real) denotes the same type.

Tuple (product) types  The type expression \( \tau_1 \ast \cdots \ast \tau_n \) denotes the type of \( n \)-tuples whose elements are resp. of type \( \tau_i \). This includes the type of pairs as well as the type of tuples of 3 elements or more. Note that \( \tau_1 \ast (\tau_2 \ast \tau_3) \), \( (\tau_1 \ast \tau_2) \ast \tau_3 \) and \( \tau_1 \ast \tau_2 \ast \tau_3 \) 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 \( \tau \), it gives the type \( \tau \text{ distr} \) of sub-distributions over \( \tau \). Note that the application is in postfix form. One other common type constructors is the one list of polymorphic list, the type expression \( \tau \text{ list} \) denoting the type of lists whose elements are of type \( \tau \).

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 \( (\tau, \sigma) \text{ map} \) (whose keys are of type \( \tau \) and values of type \( \sigma \)) 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:
Here \( t \) is the type of records with field projections \( x \) of type \( \text{int} \), and \( y \) of type \( \text{bool} \). 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:

\[
\begin{align*}
t & = \{ x : \text{int}; y : \text{bool}; \} . \\
u & = \{ y : \text{real}; yy : \text{int}; yyy : \text{real}; \} .
\end{align*}
\]

Then, a value \( v \) of type \( \text{int} \ t \) would have fields \( x \) and \( f \) of types \( \text{int} \) and \( \text{int} \to \text{int} \), respectively; and a value \( v \) of type \( (\text{int}, \text{bool}) \ u \) would have fields \( x \) and \( f \) with types \( \text{int} \) and \( \text{int} \to \text{bool} \), respectively.

Datatypes and datatype operators may be declared like this:

\[
\begin{align*}
\text{enum} & = \{ \text{First} \mid \text{Second} \mid \text{Third} \} . \\
\text{either_int_bool} & = \{ \text{First of int} \mid \text{Second of bool} \} . \\
\text{intlist} & = [ \text{Nil} \mid \text{Cons of (int } \& \text{ intlist)} ] . \\
\text{a list} & = [ \text{Nil} \mid \text{Cons of 'a } \& \text{ 'a list} ] .
\end{align*}
\]

Here, \( \text{First} \), \( \text{Second} \), \( \text{Third} \), \( \text{Nil} \) and \( \text{Cons} \) are constructors, and must be distinct from all operators, record projections and other constructors. \( \text{enum} \) is an enumerated type with the three elements \( \text{First} \), \( \text{Second} \) and \( \text{Third} \). The elements of \( \text{either_int_bool} \) consist of \( \text{First} \) applied to an integer, or \( \text{Second} \) applied to a boolean, and the datatype operator \( \text{either} \) is simply its generalization to arbitrary types \( 'a \) and \( 'b \). \( \text{intlist} \) is an inductive datatype: its elements are \( \text{Nil} \) and the results of applying \( \text{Cons} \) to a pairs of the form \( (x, ys) \), where \( x \) is an integer and \( ys \) is a previously constructed \( \text{intlist} \). Note that a vertical bar (\( | \) ) is permitted before the first constructor of a datatype. Finally, \( \text{list} \) is the generalization of \( \text{intlist} \) to lists over an arbitrary type \( 'a \), but with a twist. The use of \( \& \) means that \( \text{Cons} \) is “curried”: instead of applying \( \text{Cons} \) to a pair \( (x, ys) \), one gives it \( x : 'a \) and \( ys : 'a \text{ list} \) one at a time, as in \( \text{Cons x ys} \). 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:

\[
\begin{align*}
\text{tree} & = [ \text{Leaf of int} \mid \text{Cons of tree } \& \text{ int } \& \text{ tree} ] .
\end{align*}
\]

\( \text{Cons tr}_1 x tr_2 \) will be the tree constructed from an integer \( x \) and trees \( tr_1 \) and \( tr_2 \). \textsc{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:

\[
\begin{align*}
t & = \text{int } \& \text{ bool} . \\
('a, b) \text{ arr} & = 'a \to 'b .
\end{align*}
\]

Then, e.g., \( (\text{int} \to \text{bool}) \text{ arr} \) is the same type as \( \text{int } \to \text{ bool} \).

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

\[
\begin{align*}
t . \\
('a, b) \ u . \\
t , ('a, b) \ u .
\end{align*}
\]
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

\[ \text{fun } (x : t_1) \Rightarrow e, \]

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 \( t_1 \rightarrow t_2 \). Function application is written using juxtapositioning, so that if \( e_1 \) has type \( t_1 \rightarrow t_2 \), 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

\[ (\text{fun } x : \text{int} \Rightarrow \text{fun } y : \text{int} \Rightarrow \text{fun } z : \text{bool} \Rightarrow y) \ 0 \ 1 \ \text{false} \]

which evaluates to 1. It may be abbreviated to

\[ (\text{fun } (x y : \text{int}, z : \text{bool}) \Rightarrow y) \ 0 \ 1 \ \text{false} \]

or

\[ (\text{fun } (x : \text{int}) (y : \text{int}) (z : \text{bool}) \Rightarrow y) \ 0 \ 1 \ \text{false} \]

or (letting EasyCrypt carry out type inference)

\[ (\text{fun } x y z \Rightarrow y) \ 0 \ 1 \ \text{false} \]

In the type inference, only the type of \( y \) is determined, but that’s acceptable.

EasyCrypt has let expressions

\[ \text{let } x : t = e \text { in } e \]

which are equivalent to

\[ (\text{fun } x : t \Rightarrow e) e \]

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.,

\[ \text{op } x : \text{int } = 3. \]
\[ \text{op } f : \text{int} \rightarrow \text{bool} \rightarrow \text{int} = \text{fun } (x : \text{int}) (y : \text{bool}) \Rightarrow x. \]
\[ \text{op } g : \text{bool} \rightarrow \text{int} = f \ 1. \]
\[ \text{op } y : \text{int} = g \text{ true}. \]
\[ \text{op } z = f \ 1 \ \text{true}. \]

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

\[ \text{op } f (x : \text{int}) (y : \text{bool}) = x. \]

or

\[ \text{op } f (x : \text{int}, y : \text{bool}) = x. \]

Polymorphic operators may be declared, as in

\[ \text{op } g ['a', 'b'] : 'a \rightarrow 'b \rightarrow 'a = \text{fun } (x : 'a, y : 'b) \Rightarrow x. \]
or

\[
\text{op } g \ [\, \text{'a}, \text{'b}] (x : \text{'a}, y : \text{'b}) = x.
\]

or

\[
\text{op } g (x : \text{'a}, y : \text{'b}) = x.
\]

Here \(g\) has all the types formed by substituting types for the types variable \(\text{'a}\) and \(\text{'b}\) in \(\text{'a} \rightarrow \text{'b} \rightarrow \text{'a}\). This allows us to use \(g\) at different types

\[
\text{op } a = g \text{ true } 0.
\]

\[
\text{op } b = g \text{ 0 } \text{ false}.
\]

making \(a\) and \(b\) evaluate to \(\text{true}\) and \(0\), respectively.

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

\[
\text{op } x : \text{int}.
\]

\[
\text{op } f : \text{int} \rightarrow \text{int}.
\]

\[
\text{op } g \ [\, \text{'a}, \text{'b}] : \text{'a} \rightarrow \text{'b} \rightarrow \text{'a}.
\]

Equivalently, \(f\) and \(g\) may be declared like this:

\[
\text{op } f (x : \text{int}) : \text{int}.
\]

\[
\text{op } g \ [\, \text{'a}, \text{'b}] (x : \text{'a}, y : \text{'b}) : \text{'a}.
\]

One may declare multiple abstract operators of the same type:

\[
\text{op } f, g : \text{int} \rightarrow \text{int}.
\]

\[
\text{op } g, h \ [\, \text{'a}, \text{'b}] : \text{'a} \rightarrow \text{'b} \rightarrow \text{'a}.
\]

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

\[
\text{op } (--) \ [\, \text{'a}, \text{'b}] (x : \text{'a}) (y : \text{'b}) = x.
\]

\[
\text{op } x : \text{int} = (--) 0 \, \text{true}.
\]

\[
\text{op } x' : \text{int} = 0 \, -- \, \text{true}.
\]

\[
\text{op } y : \text{bool} = (--) \, \text{true} \, \text{0}.
\]

\[
\text{op } y' : \text{bool} = \text{true} \, -- \, 0.
\]

then \(x\) and \(x'\) evaluate to \(0\), and \(y\) and \(y'\) evaluate to \(\text{true}\).

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

\[
\text{op } x : \text{int}.
\]

\[
\text{op } f : \text{int} \rightarrow \text{int}.
\]

\[
\text{op } ! : \text{int} \rightarrow \text{int}.
\]

\[
\text{op } y : \text{int} = ! \, f \, x.
\]

\[
\text{op } y' : \text{int} = ![f \, x].
\]

then \(y\) and \(y'\) both evaluate to the result of applying the abstract operator \(!\) of type \(\text{int} \rightarrow \text{int}\) to the result of applying the abstract operator \(f\) of type \(\text{int} \rightarrow \text{int}\) to the abstract value \(x\) of type \(\text{int}\). 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).
• \([\text{[]}\] \) \[\text{[]}\] is translated to "\[\text{[]}\]". E.g., if we declare

\begin{align*}
\text{op } "[\text{[]}]" : \text{int} & = 3. \\
\text{op } x : \text{int} & = [\text{[]}].
\end{align*}

then \(x\) will evaluate to 3.

• \((\text{\_} | e)\) If \(e\) is an expression, then \(\text{\_}|e\) is translated to "\(\_|e\)" as long as the latter expression is well-typed. E.g., if we declare

\begin{align*}
\text{op } "[\text{[]}]" : \text{int} & \rightarrow \text{bool}. \\
\text{op } x : \text{bool} & = "[\text{[]}]" 3. \\
\text{op } y : \text{bool} & = \text{\_}|3|.
\end{align*}

then \(y\) will evaluate to the same value as \(x\).

• \((\text{\_}\[\text{[]}\])\) 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

\begin{align*}
\text{op } "\_\[\text{[]}\]" : \text{int} & \rightarrow \text{int} \rightarrow \text{bool}. \\
\text{op } x : \text{bool} & = "\_\[\text{[]}\]" 3 4. \\
\text{op } y : \text{bool} & = 3.[4].
\end{align*}

then \(y\) will evaluate to the same value as \(x\).

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

\begin{align*}
\text{op } "\_\[\text{<>}\]" : \text{int} & \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{bool}. \\
\text{op } x : \text{bool} & = "\_\[\text{<>}\]" 3 4 5. \\
\text{op } y : \text{bool} & = 3.[4 <- 5].
\end{align*}

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 \(e_1 :: \ldots :: e_n :: [\text{[]}]\) whenever the latter expression is well-typed. The initial argument of "\(\_\[\_\]" and "\(\_\[\_\<\text{<>}\]\) have higher precedence than even function application. E.g., one can’t omit the parentheses in

\begin{align*}
\text{op } f : \text{int} & \rightarrow \text{int}. \\
\text{op } y : \text{bool} & = (f\ 3).[4]. \\
\text{op } z : \text{bool} & = (f\ 3).[4 <- 5].
\end{align*}

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

\begin{align*}
\text{op } (=) [a] : 'a \rightarrow 'a \rightarrow \text{bool}. \\
\text{op } [!] : \text{bool} \rightarrow \text{bool}. \\
\text{op } (||) : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}. \\
\text{op } (\\slash) : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}. \\
\text{op } (&&) : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}. \\
\text{op } (\Rightarrow) : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}. \\
\text{op } (\Leftrightarrow) : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}. \\
\text{op } \mu : 'a \text{distr} \rightarrow ('a \rightarrow \text{bool}) \rightarrow \text{real}.
\end{align*}

The operator = is equality. On the booleans, we have negation \(\neg\), two forms of disjunction \((\lor\ \text{and } \text{||})\) and conjunction \((\land\ \text{and } \&\&)\), implication \((=\Rightarrow\ \text{and } \Rightarrow\) and if-and-only-if \((\Leftrightarrow)\). 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 \(e_1 \lor e_2\) is treated as \(!((e_1 \lor e_2))\). \(!\) is not an operator, but it has the precedence and non-associative status of Section 2.1. The intended meaning of \(\mu dp\) 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 \(\text{true}\)).

If \(e\) is an expression of type \(\text{int}\), then \(e \% \text{r}\) is the corresponding \(\text{real}\). \(\% \text{r}\) has higher precedence than even function application.

If \(e_1\) is an expression of type \(\text{bool}\) 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 \(\text{true}\), and evaluates to \(e_3\), if \(e_1\) evaluates to \(\text{false}\). Conditionals may also be written using if-then-else notation:

\[
\text{if } e_1 \text{ then } e_2 \text{ else } e_3
\]

E.g., if we write

\[
op x : \text{int} = (3 < 4) \ ? \ 4 + 7 : (9 - 1).
\]
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 \(\text{bool}\), \(\text{int}\) and \(\text{real}\), and the type operator \(\text{distr}\), the EASYCRYPT Library (see Chapter 5) provides corresponding theories, \(\text{Bool}\), \(\text{Int}\), \(\text{Real}\) and \(\text{Distr}\). These theories provide various operations, axioms, etc. To make use of a theory, one must “require” it. E.g.,

\[
\text{require } \text{Bool Int Real Distr}.
\]
will make the theories just mentioned available. This would allow us to write, e.g.,

\[
op x = \text{Int.}(+) \ 3 \ 4.
\]
making \(x\) evaluate to \(7\). But to be able to use \(+\) and the other operators provided by \(\text{Int}\) in infix form and without qualification (specifying which theory to find them in), we need to import \(\text{Int}\). If we do

\[
\text{import } \text{Bool Int Real}.
op x : \text{int} = 3 + 4 - 7 \ast 2.
op y : \text{real} = 5\% \text{r} \ast 3\% \text{r} / 2\% \text{r}.
op z : \text{bool} = x\% \text{r} \geq y.
\]
we’ll end up with \(z\) evaluating to \(\text{false}\). One may combine requiring and importing in one step:

\[
\text{require import } \text{Bool Int Real Distr}.
\]
We’ll cover theories and their usage in detail in Chapter 4.

Requiring the theory \(\text{Bool}\) makes available the value \(\{0,1\}\) of type \(\text{bool distr}\), 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 \(\text{Distr}\) make available syntax for the uniform distribution of integers from a finite range. If \(e_1\) and \(e_2\) are expressions of type \(\text{int}\) denoting \(n_1\) and \(n_2\), respectively, then \([e_1..e_2]\) is the value of type \(\text{int distr}\) 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 \(n_1 \gt n_2\), 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 \( x : \text{int} \ast \text{int} \ast \text{bool} = (3, 4, \text{true}) \).
op \( b : \text{bool} = x \).
type \( t = \{ u : \text{int}; v : \text{bool}; \} \).
op \( y : t = \{ \text{v} = \text{false}; u = 10; \} \).
op \( a : \text{bool} = y \).

Then, \( b \) evaluates to \( \text{true} \), and \( a \) evaluates to \( \text{false} \). 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 \( x : (\text{int, bool}) \text{either} = \text{Fst} \, 10 \).
op \( y : \text{int} \rightarrow (\text{int, bool}) \text{either} = \text{Fst} \).
op \( z : (\text{int, bool}) \text{either} = y \, 10 \).

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 \( \text{fst} \) by:

\[
\begin{align*}
op \text{fst} ['a, 'b] (\text{def : 'a}) (\text{ei : ('a, 'b) either}) : 'a = 
& \text{with} \ \text{ei = Fst a} \Rightarrow \text{a} \\
& \text{with} \ \text{ei = Snd b} \Rightarrow \text{def}.
\end{align*}
\]

\[
\begin{align*}
op l1 : (\text{int, bool}) \text{either} = \text{Fst} \, 10. 
op l2 : (\text{int, bool}) \text{either} = \text{Snd} \, \text{true}. 
op m1 : \text{int} = \text{fst} (-1) \, l1. 
op m2 : \text{int} = \text{fst} (-1) \, l2.
\end{align*}
\]

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 \( \text{length} \) that computes the length of a list:

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

Then \( \text{n} \) 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 Distr.

module M = {
  var x : int

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

  proc incr(n : int) : unit = {
    x <- x + n;
  }
}
```
$\mathcal{M}$ has one global variable—$x$—which is used by the procedures of $\mathcal{M}$—init, incr, get and main. Global variables must be declared before the procedures that use them.

The procedure $\text{init}$ ("initialize") has a parameter (or argument) $\text{bnd}$ ("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 $\text{bnd}$. The return type of $\text{init}$ is unit, whose only element is $\text{tt}$; this is implicitly returned by $\text{init}$ upon exit.

The procedure $\text{incr}$ ("increment"), increments the value of $x$ by its parameter $n$. The procedure $\text{get}$ 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

```plaintext
module $\mathcal{M}$ = {
  var x : int

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

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

  proc get() = {
    return x;
  }
}
```

Listing 2.1: Simple Module
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:

```plaintext
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

```plaintext
(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.

```plaintext
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, \texttt{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:

\begin{verbatim}
if (z <= 5) y <- y - z;
else y <- y + (z - 5);
\end{verbatim}

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

\begin{verbatim}
if (z <= 5) y <- y - z;
\end{verbatim}

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

\texttt{require import Bool Int.}

\texttt{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 M.x > 0; }

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:

\begin{verbatim}
n : int, m : int, b : bool
\end{verbatim}

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

\begin{verbatim}
proc f() : unit = {
M.x <- M.x + 1;
}
\end{verbatim}

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
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 \(X.f()\)

```plaintext
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:

```plaintext
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:

```plaintext
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:

```plaintext
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.,

```plaintext
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.

```plaintext
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 (tris > 0) {
    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.

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; }
```
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:

```ocaml
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

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

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

```ocaml
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:

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

The full Guessing Game example is contained in Listing 2.7.
var guessed : bool
proc init(secret tries : int) : unit = {
  sec <- secret;
  tris <- tries;
  guessed <- false;
}

proc guess(guess : int) : unit = {
  if (tris > 0) {
    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(O : OR) : ADV(O) = {
  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 (tries > 0) {
      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 *)
SimpAdv is a simple implementation of an adversary. The inclusion of the constraint ADV(0) in SimpAdv’s declaration

\texttt{module SimpAdv(0 : OR) : ADV(0) = ...}

makes \texttt{EasyCrypt} check that: its implementation of \texttt{chooseRange} doesn’t use \texttt{O} at all; its implementation of \texttt{doGuessing} doesn’t use any of \texttt{O}’s procedures other than \texttt{guess}; and that \texttt{chooseRange} initializes SimpAdv’s global variables. Its \texttt{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 \texttt{Game} to see why 10 is a sensible choice). The \texttt{doGuessing} procedure makes 10 random guesses.

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

\texttt{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:

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

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

- It calls \texttt{A}’s \texttt{chooseRange} procedure to get the adversary’s choice of guessing range. If the range doesn’t have at least ten elements, it returns \texttt{false} without doing anything else—the adversary has supplied a range that’s too small.

- Otherwise, it uses \texttt{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 \texttt{A.doGuessing}, allowing the adversary to attempt the guess the secret.

- Finally, it calls \texttt{Or.guessed} to learn whether the adversary has guessed the secret, returning this boolean value as its result.

Finally, the declaration

\texttt{module SimpGame = Game(SimpAdv)}.
CHAPTER 2. SPECIFICATIONS

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:

```python
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
\[ \text{op} [!] : \text{bool} \rightarrow \text{bool}. \]

the two semantically equivalent disjunctions
\[ \text{op} (||) : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}. \]
\[ \text{op} (\lor) : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}. \]

the two semantically equivalent conjunctions
\[ \text{op} (\&\&) : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}. \]
\[ \text{op} (\&\&) : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}. \]

implication
\[ \text{op} (\Rightarrow) : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}. \]

and if-and-only-if
\[ \text{op} (\Leftrightarrow) : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}. \]

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
\[ \text{forall } (x : t), \phi \]

and
\[ \text{exists } (x : t), \phi \]

respectively, where the formula \( \phi \) 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.,
\[ \text{forall } (x : \text{int}, y : \text{int}, z : \text{bool}), ... \]
\[ \text{forall } (x : \text{int}, z : \text{bool}), ... \]
\[ \text{forall } (x : \text{int}) (z : \text{bool}), ... \]
\[ \text{exists } (x : \text{int}, y : \text{int}, z : \text{bool}), ... \]
\[ \text{exists } (x : \text{int}, z : \text{bool}), ... \]
\[ \text{exists } (x : \text{int}) (z : \text{bool}), ... \]

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:
\[ \text{pred } P0. \]
\[ \text{pred } P1 : \text{int}. \]
\[ \text{pred } P2 : \text{int} \& (\text{int} \& \text{bool}). \]
\[ \text{pred } P3 : \text{int} \& (\text{int} \& \text{bool}) \& (\text{real} \rightarrow \text{int}). \]

\( P0, P1, P2 \) and \( P3 \) are extended expressions of types \( \text{bool, int} \rightarrow \text{bool, int} \rightarrow \text{int} \& \text{bool} \rightarrow \text{bool} \)
and \( \text{int} \rightarrow \text{int} \& \text{bool} \rightarrow (\text{real} \rightarrow \text{int}) \rightarrow \text{bool} \), respectively. The parentheses are mandatory in \( (\text{int} \& \text{bool}) \) and \( (\text{real} \rightarrow \text{int}) \). Thus, if \( e_1, e_2 \) and \( e_3 \) are extended expressions of types \( \text{int, int} \& \text{bool} \) and \( \text{real} \rightarrow \text{int} \), respectively, then \( P0, P1e_1, P2e_1e_2 \) and \( P3e_1e_2e_3 \) are formulas.

Concrete predicates are defined in a way that is similar to how operators are declared. E.g., if we declare
CHAPTER 2. SPECIFICATIONS

28

pred \( Q (x \; y : \text{int}, \; z : \text{bool}) = x = y \land z. \)

or

\[
\text{pred } Q (x : \text{int}) (y : \text{int}) (z : \text{bool}) = x = y \land z.
\]

then \( Q \) is an extended expression of type

\[
\text{int} \rightarrow \text{int} \rightarrow \text{bool} \rightarrow \text{bool}
\]

meaning that, e.g.,

\[
(\text{fun} \; (b : \text{bool} \rightarrow \text{bool}) \Rightarrow b \; \text{true}) \; (Q \; 3 \; 4)
\]

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

\[
\text{pred } R \left[ \begin{array}{c}
\mathit{a}, \mathit{b} \end{array} \right] : \left( \begin{array}{c}
\mathit{a}, \mathit{a} \ast \mathit{b} \end{array} \right).
\]

\[
\text{pred } R \left[ \begin{array}{c}
\mathit{a}, \mathit{b} \end{array} \right] (x : \mathit{a}, \; y : \mathit{a} \ast \mathit{b}) = (y.\mathit{1} = x).
\]

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:

\[
\forall \&m, \; \phi
\]

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:

\[
\text{module } X = \{ \text{var } x : \text{int } \}. \nomorelines
\text{module } Y = \{ \text{var } y : \text{int } \}.
\]

Then, this is a (true) formula:

\[
\forall \&m, \; X.x{\{m\}} < Y.y{\{m\}} \Rightarrow X.x{\{m\}} + 1 <= Y.y{\{m\}}
\]

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

• \text{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 \( \text{(glob } M \text{)}(m) \) is the value of type \text{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 \( \phi \) is a formula and \&m is a memory, then \( \phi{\{m\}} \) means the formula in which every subterm \( u \) of \( \phi \) consisting of a variable or \text{res} or \text{glob } M, for a module \( M \), is replaced by \( u{\{m\}} \). For example,

\[
(Y1.y = Y1.z \Rightarrow Y1.z = Y1.y){\{m\}}
\]

expands to

\[
Y1.y{\{m\}} = Y1.z{\{m\}} \Rightarrow Y1.z{\{m\}} = Y1.y{\{m\}}
\]

The parentheses are necessary, because \( \{m\} \) has higher precedence than even function application. We say that \&m satisfies \( \phi \) 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
forall \((M <: T), \phi\)

means

for all modules \(M\) satisfying \(T\), \(\phi\) holds.

Formulas can’t talk about module equality.

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

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

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 \(N_i\), \(\phi\) 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

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

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 \(\phi\) may involve the term \(\text{res}\), whose type is \(M.p\)’s return type, as well as global variables of modules.

Occurrences in \(\phi\) 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 \(\phi\). 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 \(\text{res}\) 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

\[
\text{hoare}[M.p : \phi \Rightarrow \psi]
\]

where:

– \(p\) is a procedure of module \(M\);
– the formula \(\phi\) may involve the global variables of declared modules, as well as the parameters of \(M.p\);
the formula \( \psi \) may involve the term \( \text{res} \), whose type is \( M \).\( p \)'s return type, as well as the global variables of declared modules.

Occurrences in \( \phi \) and \( \psi \) 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 \( \text{HL} \) judgment is that, for all initial memories \( \&m \) satisfying \( \phi \) 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 \( \text{res} \) and the global variables of declared modules satisfies \( \psi \).

• (\( \text{pHL} \) Judgments) A \( \text{pHL} \) judgment has one of the forms

\[
\begin{align*}
\text{phoare} \ [M . p : \phi \Rightarrow \psi] < e \\
\text{phoare} \ [M . p : \phi \Rightarrow \psi] = e \\
\text{phoare} \ [M . p : \phi \Rightarrow \psi] > e
\end{align*}
\]

where:

– \( p \) is a procedure of module \( M \);
– the formula \( \phi \) may involve the global variables of declared modules, as well as the parameters of \( M . p \);
– the formula \( \psi \) may involve the term \( \text{res} \), whose type is \( M . p \)'s return type, as well as the global variables of declared modules;
– \( e \) is an expression of type \( \text{real} \).

Occurrences in \( \phi \) and \( \psi \) 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 \( \text{pHL} \) judgment is that, for all initial memories \( \&m \) satisfying \( \phi \) 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 \( \text{res} \) and the global variables of declared modules satisfies \( \psi \)

has the indicated relation to the value of \( e \).

• (\( \text{pRHL} \) Judgments) A \( \text{pRHL} \) judgment has the form

\[
\text{equiv}[M . p - N . q : \phi \Rightarrow \psi]
\]

where:

– \( p \) is a procedure of module \( M \), and \( q \) is a procedure of module \( N \);
– the formula \( \phi \) 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 \( \psi \) may involve the global variables of declared modules, \( \text{res}(1) \), which has the type of \( M . p \)'s return type, and \( \text{res}(2) \), which has the type of \( N . q \)'s return type.
Occurrences in $\psi$ 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 $\phi$ and $\psi$ 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 $\phi$ holds, then the sub-distributions on memories $\Pi_p$ and $\Pi_q$ 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 $\psi$, in the following sense. (The probability of a memory in $\Pi_p$ (resp., $\Pi_q$) is the probability that $p$ (resp., $q$) will terminate with that memory. $\Pi_p$ and $\Pi_q$ are sub-distributions on memories because $p$ and $q$ may fail to terminate.)

We say that $(\Pi_p, \Pi_q)$ satisfy $\psi$ iff there is a function $f$ dividing the probability assigned to each memory $&m$ by $\Pi_p$ among the memories $&n$ related to it by $\psi$ ($&m$ and $&n$ are related according to $\psi$ iff $\psi$ 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 $\Pi_q$ is the sum of all the probabilities distributed to $&n$ by $f$. (When $\psi$ is an equivalence like $\{\text{res}\}$ (i.e., $\text{res}_1 = \text{res}_2$), this is particularly easy to interpret.)

- (Lossless Assertions) A lossless assertion has the form

$$\text{islossless } M.p$$

and is simply an abbreviation for

$$\text{phoare } [M.p : \text{true} \implies \text{true}] = 1/r$$

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

```plaintext
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

$$\Pr[\text{G1.f()} @ &m : \text{res}]$$

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

\[ \text{boare} [G2.f : \text{true } \rightarrow \text{!res}] \]

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

• The pHL judgement

\[ \text{phoare} [G2.f : \text{true } \rightarrow \text{res}] = (1\%r / 2\%r) \]

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

• The pRHL judgement

\[ \text{equiv} [G1.f \sim G2.f : \text{true } \rightarrow \text{res}] \]

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

• The lossless assertion

\[ \text{lossless} G2.f \]

says that \( G2.f() \) always terminates, no matter what memory it’s run in. (This judgement is \text{true}.)

### 2.5.2 Axioms and Lemmas

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

\begin{align*}
\text{axiom} & \quad \text{Sym} : \forall (x \ y : \text{int}), x = y \Rightarrow y = x. \\
\text{lemma} & \quad \text{Sym} : \forall (x \ y : \text{int}), x = y \Rightarrow y = x.
\end{align*}

The difference between axioms and lemmas is that axioms are trusted by \text{EASYCRYPT}, whereas lemmas must be proved, in the steps that follow. The \text{proof} of a lemma has the form

\begin{align*}
\text{proof}. \\
\text{tactic}_1. \ldots \text{tactic}_n. \\
\text{qed}.
\end{align*}

Actually the \text{proof} step is optional, but it’s good style to include it. The steps of the proof consist of tactic applications; but \text{print} and \text{search} commands are also legal steps. The \text{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 \text{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:

\begin{align*}
\text{lemma} & \quad \text{name} : \phi \text{ by } [\text{tactic}]. \\
\text{or} \\
\text{lemma} & \quad \text{name} : \phi \text{ by } [].
\end{align*}

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

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

\begin{align*}
\text{lemma} & \quad \text{Sym} (x : \text{int}) (y : \text{int}) : x = y \Rightarrow y = x. \\
\text{or} \\
\text{lemma} & \quad \text{Sym} (x \ y : \text{int}) : x = y \Rightarrow y = x.
\end{align*}
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:

```plaintext
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

```plaintext
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:

```plaintext
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

```plaintext
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:

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

Then lemma Lossless

```plaintext
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

```plaintext
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

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

this is a goal:
\begin{center}
\texttt{Type variables: <none>}
\end{center}
\begin{center}
\texttt{X : T\{G\}}
\texttt{n : int}
\end{center}
\begin{center}
\texttt{LL: islossless X.f}
\end{center}
\begin{center}
\texttt{pre = G.x = n}
\texttt{G(X).g}
\texttt{[=] 1\%r}
\end{center}
\begin{center}
\texttt{post = G.x = n}
\end{center}

The conclusion of this goal is just a nonlinear rendering of the formula
\texttt{\texttt{phoarse \linebreak [G(X).g : G.x = n ==> G.x = n] = 1\%r.}}

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

Internally, \textsc{EasyCrypt}'s proof engine also works with \texttt{pRHL}, \texttt{pHL} and \texttt{HL} judgments involving lists of statements rather than procedure names, which we'll call \textit{statement judgements}, below. For example, given this declaration
\begin{center}
\texttt{module M = { proc f(x : int) = {}
\texttt{if (x \%\% 3 = 1) x = x + 4;}
\texttt{else x = x + 2;}
\texttt{return x;}
\}}.}
\end{center}

this is an \texttt{pHL} statement judgement:
\begin{center}
\texttt{Type variables: <none>}
\end{center}
\begin{center}
\texttt{x : int}
\end{center}
\begin{center}
\texttt{zor1_x : x = 1 \lor x = 2}
\end{center}
\begin{center}
\texttt{Context : M.f}
\end{center}
\begin{center}
\texttt{pre = x \%\% 3 = x}
\end{center}
\begin{center}
\texttt{(1--) if (x \%\% 3 = 1) { (1.1) x <- x + 4}
\texttt{(1--) } else { (1?1) x <- x + 2
\texttt{(1--) }
\end{center}
\begin{center}
\texttt{post = x \%\% 3 = x \%\% 2 + 1}
\end{center}

The pre- and post-conditions of a statement judgement may refer to the parameters and local variables of the \textit{procedure context} of the conclusion—\texttt{M.f} in the preceding example. They may also refer to the memories \texttt{\&1} and \texttt{\&2} in the case of \texttt{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

\[
\text{hoare}\{\text{if } (x \mod 3 = 1) \{ \cdots \} : x \mod 3 = n \Rightarrow x \mod 3 = n \mod (2 + 1)\}
\]

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

\[
\text{lemma } \text{PairEq }\{\text{a}, \text{b}\} : \\
\text{forall } (x x': \text{a}) (y y': \text{b}), \\
x = x' \Rightarrow y = y' \Rightarrow (x, y) = (x', y').
\]

gives rise to the goal

Type variables: 'a, 'b

\[
\text{forall } (x x': \text{a}) (y y': \text{b}), x = x' \Rightarrow y = y' \Rightarrow (x, y) = (x', y')
\]

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

\[
\text{lemma } \text{PairEq }\{x x': \text{a}) (y y': \text{b}) : \\
x = x' \Rightarrow y = y' \Rightarrow (x, y) = (x', y').
\]

gives rise to

Type variables: 'a, 'b

x : 'a
x' : 'a
y : 'b
y' : 'b

\[
x = x' \Rightarrow y = y' \Rightarrow (x, y) = (x', y')
\]

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 \lor x = 2

Context : \text{M.f}

pre = x \mod 3 = x

(1--) \text{if } (x \mod 3 = 1) \{
(1.1) x \leftarrow x + 4
(1--) \} \text{ else } \{
(1?1) x \leftarrow x + 2
(1--) \}

post = x \mod 3 = x \mod (2 + 1)
then applying the \texttt{if} tactic (handle a conditional) reduces (replaces) this goal with the two goals

\begin{verbatim}
Type variables: <none>

x : int
zor1_x: x = 1 \lor x = 2

Context : M.f
pre = x \mod 3 = x \land x \mod 3 = 1
(1) x <- x + 4
post = x \mod 3 = x \mod 2 + 1
\end{verbatim}

and

\begin{verbatim}
Type variables: <none>

x : int
zor1_x: x = 1 \lor x = 2

Context : M.f
pre = x \mod 3 = x \land x \mod 3 <> 1
(1) x <- x + 2
post = x \mod 3 = x \mod 2 + 1
\end{verbatim}

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

\begin{verbatim}
Type variables: <none>

x : int
zor1_x: x = 1 \lor x = 2

forall \&hr, x \mod 3 = x \land x(\&hr) \mod 3 = 1 => (x(\&hr) + 4) \mod 3 = x \mod 2 + 1
\end{verbatim}

then applying the \texttt{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 \texttt{qed}, when the list of goals becomes empty. And this must be done before anything else may be done.

\textbf{Remark.} In the descriptions of \texttt{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.

\section{Ambient logic}

In this section, we describe the proof terms, tactics and tacticals of \texttt{EasyCrypt}’s ambient logic.

\subsection{Proof Terms}

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

\begin{verbatim}
forall (x y : bool), x = y => forall (z : bool), y = z => x = z.
\end{verbatim}
CHAPTER 3. TACTICS

\[ p ::= \_ \quad \text{proof hole} \]
\[ (X, q_1, \ldots, q_n) \quad \text{lemma application} \]
\[ q ::= e \quad \text{expression} \]
\[ p \quad \text{proof term} \]

Figure 3.1: Proof Terms

Introduces the assumptions

\[
\begin{align*}
x & : \text{bool} \\
y & : \text{bool} \\
eq_{xy} & : x = y \\
z & : \text{bool} \\
eq_{yz} & : y = z
\end{align*}
\]

(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 \( x : \text{bool} \).

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 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. 59) tactic.

\textbf{Fixme Note: Need explanation of how a proof term may be used in forward reasoning.}

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

\[
\begin{align*}
pred P : \text{int}. \\
pred Q : \text{int}. \\
pred R : \text{int}. \\
axiom P (x : \text{int}) & : \text{P} x. \\
axiom Q (x : \text{int}) & : \text{P} x \Rightarrow \text{Q} x. \\
axiom R (x : \text{int}) & : \text{P}(x + 1) \Rightarrow \text{Q} x \Rightarrow \text{R} x.
\end{align*}
\]

Then, given that \( x : \text{int} \) is an assumption,

\[
(R \times (P(x + 1)) \times (Q \times (P x)))
\]

is a proof term proving the conclusion \( R \times \). And

\[
(R \times \_ \times (Q \times \_))
\]

is a proof term that turns proofs of \( P(x + 1) \) and \( P \times \) into proofs of \( R \times \). When used in backward reasoning, it will reduce a goal with conclusion \( R \times \) to subgoals with conclusions \( P(x + 1) \) and \( P \times \).

\textbf{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 \( \_ \) for \( (R \times \_ \_ \_) \). When used in backward reasoning, it will reduce a goal with conclusion \( R \times \) to subgoals with conclusions \( P(x + 1) \) and \( Q \times \). \textbf{Fixme Note: In forward reasoning they aren’t equivalent—why?}

3.2.2 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 \( i_1 \) through \( i_n \) 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.

### 3.2.3 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 3.2.2.

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

---

**Figure 3.2: Introduction Patterns**

<table>
<thead>
<tr>
<th>(\iota)</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b)</td>
<td>name</td>
</tr>
<tr>
<td>(!)</td>
<td>rename</td>
</tr>
<tr>
<td>(-)</td>
<td>no name</td>
</tr>
<tr>
<td>(+)</td>
<td>auto revert</td>
</tr>
<tr>
<td>(?)</td>
<td>find name</td>
</tr>
<tr>
<td>(occ)</td>
<td>rewrite using assumption</td>
</tr>
<tr>
<td>(occ)</td>
<td>rewrite in reverse</td>
</tr>
<tr>
<td>(\Rightarrow)</td>
<td>substitute using assumption</td>
</tr>
<tr>
<td>(\Leftarrow)</td>
<td>substitute in reverse</td>
</tr>
<tr>
<td>(\Rightarrow)</td>
<td>replace assumption by applying proof term</td>
</tr>
<tr>
<td>(\Leftarrow)</td>
<td>substitute in reverse</td>
</tr>
<tr>
<td>({/})</td>
<td>simplify</td>
</tr>
<tr>
<td>(\text{trivial})</td>
<td>simplify then trivial</td>
</tr>
<tr>
<td>(dir)</td>
<td>unfold definition of operator</td>
</tr>
<tr>
<td>(occ)</td>
<td>clear introduced assumptions</td>
</tr>
<tr>
<td>({a_1 \ldots a_n})</td>
<td>clear introduced assumptions</td>
</tr>
<tr>
<td>(\text{/=})</td>
<td>simplify</td>
</tr>
<tr>
<td>(\text{/=})</td>
<td>simplify then trivial</td>
</tr>
<tr>
<td>(\text{dir occ} @/op)</td>
<td>unfold definition of operator</td>
</tr>
<tr>
<td>({\iota_{11} \ldots \iota_{1m_1}</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

| \(b\)    | identifier               |
| \(M\)    | module name              |
| \(km\)   | memory name              |

---

RUNS THE TACTIC \(\tau\), 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 \(\iota_{k}\) 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 \(\iota_{k}\) 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 vizz a vizz the indices \(i\)) as the tactical’s result.
• Otherwise, $\tau_k$ is a case pattern $[t_{i_1} \cdots t_{i_{m_1}} | \cdots | t_{r_1} \cdots t_{r_{m_r}}]$.

• If $\tau$ is not equivalent to $\text{idtac}$ (p. 47), the tactic fails unless $r = l'$, in which case each $G'_i$ is matched against

$$t_{i_1} \cdots t_{i_{m}} t_{k+1} \cdots t_{n}$$

by the procedure described below, with the resulting subgoals being collected into the tactical’s result.

• Otherwise, $\tau$ is equivalent to $\text{idtac}$ (p. 47) (and so $l' = 1$). In this case $G'_i$ is matched against $t_{k}, \ldots, t_{n}$ 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 $t_1, \ldots, t_n$, 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 $\text{EASYCRYPT}$ chooses the name by which the assumption is introduced (using universally quantified names as assumption bases).

• (occ $\rightarrow$) 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 $\leftarrow$) The same as the preceding case, except the rewriting is from right-to-left.

• ($\Rightarrow$) The same as $\rightarrow$, except the consumed equality assumption is used to perform a left-to-right substitution in the entire goal, i.e., in its assumptions, as well as its conclusion.

• ($\Leftarrow$) 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.

• (/s) Apply $\text{simplify}$ (p. 55) to goal’s conclusion.

• (/f) Apply $\text{trivial}$ (p. 55) to goal’s conclusion; this may solve the goal, i.e., so that the procedure’s current branch yields no goals.

• (/r) Apply $\text{simplify}$ (p. 55) and then $\text{trivial}$ (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 $\text{rewrite}$ (p. 61) tactic for the details.
• \([\{i_1 \cdots i_{m_1} \mid \cdots \mid i_r \cdots i_{m_r}\}]\)
  
  – If \(r = 0\), then the top assumption of the goal is destructed using the case (p. 63) tactic, the resulting goals are matched against \(i_2, \ldots, i_m\), and their subgoals are assembled into a list of goals.
  
  – Otherwise \(r > 0\). The goal’s top assumption is destructed using the case (p. 63) tactic, yielding subgoals \(H_1, \ldots, H_p\). If \(p \neq r\), the procedure fails. Otherwise each subgoal \(H_i\) is matched against

\[i_{i_1} \cdots i_{i_m}, i_2 \cdots i_n\]

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:

```
```

produces

```
Type variables: <none>

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

x = z
```

Or we can use the ! pattern suffix to specify our own base assumption names: running

```
move=> x! x! eq! x! eq!.
```

produces
To see how the $\rightarrow$ 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=> \rightarrow.
```

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 $\rightarrow\rightarrow$ substitution pattern:

```
Type variables: <none>

x : int
y : int
H1: x = y
z : int

y = z => x = z
```

then running

```
move=> \rightarrow\rightarrow.
```

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
Then running

\texttt{move=> /Sym.}

produces

Type variables: <none>

\begin{align*}
\text{Sym: } \forall (u \ v : \text{int}), \ u = v \Rightarrow v = u \\
x : \text{int} \\
y : \text{int} \\
x = y \Rightarrow \forall (z : \text{int}), \ y = z \Rightarrow x = z
\end{align*}

And then running

\texttt{move=> ->.}

on this goal produces

Type variables: <none>

\begin{align*}
\text{Sym: } \forall (u \ v : \text{int}), \ u = v \Rightarrow v = u \\
x : \text{int} \\
y : \text{int} \\
y = x \Rightarrow \forall (z : \text{int}), \ y = z \Rightarrow x = z
\end{align*}

Finally, let’s see examples of how a disjunction assumption may be destructed, either using the \texttt{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>

\begin{align*}
x : \text{bool} \\
y : \text{bool} \\
x \lor y => \\
(x => \forall (z : \text{bool}), \ P x z) => \\
(y => \forall (z w : \text{bool}), \ Q y z w) => \\
(\forall (z : \text{bool}), \ P x z) \lor \forall (z w : \text{bool}), \ Q y z w
\end{align*}

then running

\texttt{move=> [Hx HP _ | Hy _ HQ].}

produces the two goals

Type variables: <none>

\begin{align*}
x : \text{bool} \\
y : \text{bool} \\
Hx: x \\
HP: x => \forall (z : \text{bool}), \ P x z
\end{align*}
CHAPTER 3. TACTICS

_ : y => forall (z w : bool), Q y z w

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

HP: x => forall (z : bool), P x z

(forall (z : bool), P x z) \forall 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 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.

\[ \forall: \pi_1 \cdots \pi_n \]

Generalize the patterns \(\pi_1, \ldots, \pi_n\), starting from \(\pi_n\) and going back, and then run tactic \(\tau\). This tactical is only applicable to certain tactics: move (p. 47), case (p. 63) (just the version that destructs the top assumption of a goal’s conclusion) and elim (p. 67).
• When $\pi$ 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  
```

then running

```
move: x \eq_{xy}.
```

produces

```
Type variables: <none>  
x : int  
y : int  
\forall (x : int), x = y \imp y = x  
```

In this example, one can’t generalize $x$ without also generalizing $\eq_{xy}$.

• $\pi$ 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 appropriate type.

For example, if the current goal is

```
Type variables: <none>  
x : int  
y : int  
x = y \imp y = x  
```

then running

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

produces

```
Type variables: <none>  
x : int  
y : int  
\forall (x : \bool), x = y \imp x  
```

Alternatively, running

```
m\ove: \{2\} y \{2\} x.
```

produces
3.2.4 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 3.2.3.

- **clear a₁ ··· aₙ**
  
  Clear assumptions a₁ ··· aₙ from the goal’s context. Fail if any remaining hypotheses depend on any of the aᵢ.
  
  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

\[ \text{assumption.} \]

solves the goal.

\hline
\text{reflexivity} \\
Solve goals with conclusions of the form \( b = b \) (up to computation). For example, if the current goal is

\[
\begin{align*}
\text{Type variables: } &\text{<none>} \\
y : \text{bool} \\
(\text{fun} \ (x : \text{bool}) \Rightarrow \lnot x) \ y = \lnot y
\end{align*}
\]

then running

\[ \text{reflexivity.} \]

solves the goal.

\hline
\text{left} \\
Reduce a goal whose conclusion is a disjunction to one whose conclusion is its left member. For example, if the current goal is

\[
\begin{align*}
\text{Type variables: } &\text{<none>} \\
x : \text{int} \\
y : \text{int} \\
eq_{xy} : x - 1 = y \\
x = y + 1 \lor y = x + 1
\end{align*}
\]

then running

\[ \text{left.} \]

produces the goal

\[
\begin{align*}
\text{Type variables: } &\text{<none>} \\
x : \text{int} \\
y : \text{int} \\
eq_{xy} : x - 1 = y \\
x = y + 1
\end{align*}
\]

\hline
\text{right} \\
Reduce a goal whose conclusion is a disjunction to one whose conclusion is its right member. For example, if the current goal is

\[
\begin{align*}
\text{Type variables: } &\text{<none>} \\
x : \text{int} \\
y : \text{int} \\
eq_{yz} : y - 1 = x \\
x = y + 1 \lor y = x + 1
\end{align*}
\]

then running
produces the goal

Type variables: <none>

\[
\begin{align*}
  x & : \text{int} \\
  y & : \text{int} \\
  \text{eq}_{yz} & : y - 1 = x \\
  y & = x + 1
\end{align*}
\]

If we replace \( \lor \) by \( \mid\mid \) in this example, we can see the difference between the two versions of disjunction: if the current goal is

Type variables: <none>

\[
\begin{align*}
  x & : \text{int} \\
  y & : \text{int} \\
  \text{eq}_{yz} & : y - 1 = x \\
  x & = y + 1 \mid\mid y = x + 1
\end{align*}
\]

then running

\[
\text{right}.
\]

produces the goal

Type variables: <none>

\[
\begin{align*}
  x & : \text{int} \\
  y & : \text{int} \\
  \text{eq}_{yz} & : y - 1 = x \\
  x & < \, y + 1 \Rightarrow y = x + 1
\end{align*}
\]

\[\exists e\]

Reduces proving an existential to proving the witness \( e \) satisfies the existential’s body. For example, if the current goal is

Type variables: <none>

\[
\begin{align*}
  x & : \text{int} \\
  \text{rng}_x & : 0 < x < 5 \\
  \exists e & (x0 : \text{int}), 5 < x0 < 10
\end{align*}
\]

then running

\[\exists (x + 5)\]

produces the goal

Type variables: <none>

\[
\begin{align*}
  x & : \text{int} \\
  \text{rng}_x & : 0 < x < 5 \\
  5 & < x + 5 < 10
\end{align*}
\]
CHAPTER 3. TACTICS

50

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 
  \texttt{true} or provable by \texttt{reflexivity},
- replace a logical equivalence by the direct and indirect implications,
- replace a goal of the form \( \phi_1 \land \phi_2 \) by the two subgoals for \( \phi_1 \) and \( \phi_2 \). The same applies for a goal of the form \( \phi_1 \land \phi_2 \),
- replace an equality between \( n \)-tuples by \( n \) equalities on their components.

For example, if the current goal is

\[
\begin{align*}
x : \text{int} \\
y : \text{int} \\
x = y &\iff x - 1 = y - 1
\end{align*}
\]

then running

\texttt{split}.

produces the goals

\[
\begin{align*}
x : \text{int} \\
y : \text{int} \\
x = y &\implies x - 1 = y - 1
\end{align*}
\]

\[
\begin{align*}
x : \text{int} \\
y : \text{int} \\
x - 1 = y - 1 &\implies x = y
\end{align*}
\]

And if the current goal is

\[
\begin{align*}
x : \text{int} \\
y : \text{int} \\
z : \text{int} \\
w : \text{int} \\
eq_x_y : x = y \\
eq_z_w : z = w \\
x - 1 = y - 1 \land z + 1 = w + 1
\end{align*}
\]

then running

\texttt{split}.

produces the goals
CHAPTER 3. TACTICS

Type variables: <none>

\[
x : \text{int} \\
y : \text{int} \\
z : \text{int} \\
w : \text{int} \\
eq_{xy} : x = y \\
eq_{zw} : z = w \\
\]

\[x - 1 = y - 1\]

and

Type variables: <none>

\[
x : \text{int} \\
y : \text{int} \\
z : \text{int} \\
w : \text{int} \\
eq_{xy} : x = y \\
eq_{zw} : z = w \\
\]

\[z + 1 = w + 1\]

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

Type variables: <none>

\[
x : \text{int} \\
y : \text{int} \\
z : \text{int} \\
w : \text{int} \\
eq_{xy} : x = y \\
eq_{zw} : z = w \\
\]

\[x - 1 = y - 1 \&\& z + 1 = w + 1\]

then running

\textit{split}.

produces the goals

Type variables: <none>

\[
x : \text{int} \\
y : \text{int} \\
z : \text{int} \\
w : \text{int} \\
eq_{xy} : x = y \\
eq_{zw} : z = w \\
\]

\[x - 1 = y - 1 \&\& z + 1 = w + 1\]

and

Type variables: <none>

\[
x : \text{int} \\
y : \text{int} \\
z : \text{int} \\
w : \text{int} \\
\]
CHAPTER 3. TACTICS

eq_xy: x = y
eq_zw: z = w

\[ x - 1 = y - 1 \Rightarrow z + 1 = w + 1 \]

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

\[
\text{Type variables: } \langle \text{none} \rangle
\]

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

\[\text{split.}\]

produces the goals

\[
\text{Type variables: } \langle \text{none} \rangle
\]

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

\[x - 7 = z + 2\]

and

\[
\text{Type variables: } \langle \text{none} \rangle
\]

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

\[2 + y = w - 10\]

\[\text{congr}\]

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 \textit{reflexivity} are automatically closed. Also works when the operator is used in infix form.

For example, if the current goal is

\[
\text{Type variables: } \langle \text{none} \rangle
\]

x : int
y : int
x' : int
y' : int
f : int \rightarrow \text{int} \rightarrow \text{int}
CHAPTER 3. TACTICS 53

\[ f(x + 1) (y - 1) = f(x' - 1) (y' + 1) \]
then running

\texttt{congr.}

produces the goals

Type variables: <none>

\[
\begin{align*}
  x &: \text{int} \\
  y &: \text{int} \\
  x' &: \text{int} \\
  y' &: \text{int} \\
  f &: \text{int} \rightarrow \text{int} \rightarrow \text{int} \\
  eq_{xx'} &: x = x' - 2 \\
  eq_{yy'} &: y = y' + 2 \\
  x + 1 &= x' - 1 \\
  y - 1 &= y' + 1
\end{align*}
\]

And if the current goal is

Type variables: <none>

\[
\begin{align*}
  x &: \text{int} \\
  y &: \text{int} \\
  x' &: \text{int} \\
  y' &: \text{int} \\
  eq_{xx'} &: x = x' - 2 \\
  eq_{yy'} &: y = y' + 2 \\
  x + 1 + (y - 1) &= x' - 1 + (y' + 1)
\end{align*}
\]
then running

\texttt{congr.}

produces this same pair of subgoals.

\texttt{subst x | subst}

Syntax: \texttt{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
CHAPTER 3. TACTICS

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

\texttt{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

\texttt{subst y.}

takes us to

Type variables: <none>

z : bool
w : bool
eq_zw: z = w

\[ z = w \]

from which running

\texttt{subst z.}

takes us to

Type variables: <none>

w : bool

\[ w = w \]

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

Type variables: <none>

x : bool
y : bool
z : bool
\( w : \text{bool} \)
\( \text{eq}_{yx}: y = x \)
\( \text{eq}_{yz}: y = z \)
\( \text{eq}_{zw}: z = w \)

\[
x = w
\]

then running
\[
\text{subst}.
\]
takes us to

Type variables: <none>

\[
x : \text{bool}
\]
\[
x = x
\]

\[\text{trivial}\]

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>

\[
\forall (x y : \text{int}), x = y \Rightarrow y - 1 = x - 1
\]

then running
\[
\text{trivial}.
\]
solves the goal. On the other hand, if the current goal is

Type variables: <none>

\[
\forall (x y z : \text{bool}), (x \land y \Rightarrow z) \Leftrightarrow (x \Rightarrow y \Rightarrow z)
\]

then running
\[
\text{trivial}.
\]
leaves the goal unchanged.

\[\text{FiXme Note: Be a bit more detailed about what this tactic does?}\]

\[\text{done}\]

Apply \text{trivial} (p. 55) and fail if the goal is not closed.

\[\text{simplify} \mid \text{simplify}_{x_1 \cdots x_n} \mid \text{simplify delta}\]

Reduce the goal’s conclusion to its \(\beta\iota\zeta\Lambda\)-head normal-form, followed by one step of parallel, strong \(\delta\)-reduction if \text{delta} is given. The \(\delta\)-reduction can be restricted to a set of defined symbols by replacing \text{delta} by a non-empty sequence of targeted symbols. You can reduce the conclusion to its \(\beta\)-head normal form (resp. \(\iota\), \(\zeta\), \(\Lambda\)-head normal form) by using the tactic \text{beta} (resp. \text{iota}, \text{zeta}, \text{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>

\( y : \text{bool} \)
\( z : \text{bool} \)
\( \text{imp}_yz : y \Rightarrow z \)

\( \text{true} \Rightarrow \text{true} \Leftrightarrow (y \Rightarrow \text{true} \land z) \)

then running

\textit{simplify.}

produces the goal

Type variables: <none>

\( y : \text{bool} \)
\( z : \text{bool} \)
\( \text{imp}_yz : y \Rightarrow z \)

\( y \Rightarrow z \)

And if the current goal is

Type variables: <none>

\( y : \text{bool} \)
\( z : \text{bool} \)
\( \text{imp}_yz : y \Rightarrow z \)

\( \text{false} \Leftrightarrow \text{false} \Leftrightarrow (y \Rightarrow \text{false} \land z) \)

then running

\textit{simplify.}

produces the goal

Type variables: <none>

\( y : \text{bool} \)
\( z : \text{bool} \)
\( \text{imp}_yz : y \Rightarrow z \)

\( \text{true} \)

\textbf{Fixme} Note: Is this the right place to define “convertible”? 

\( \text{-progress} \mid \text{progress } \tau \)

Break the goal into multiple \textit{simpler} ones by repeatedly applying \textit{split}, \textit{subst} and \textit{move}\Rightarrow. The tactic \( \tau \) given to \textit{progress} is tentatively applied after each step.

For example, if the current goal is

Type variables: <none>

\( \forall (x \ y \ z : \text{bool}), (x \land y \Rightarrow z) \Leftrightarrow (x \Rightarrow y \Rightarrow z) \)

then running

\textit{progress.}
solves the goal.

**Fixme** Note: Describe progress options.

<table>
<thead>
<tr>
<th>smt smt-options</th>
</tr>
</thead>
</table>

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

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

then running

```plaintext
smt.
```

solves the goal.

**Options**

- `timeout=n`: set the timeout for provers to `n` (in seconds).
- `maxprovers=n`: set the maximum number of prover running in parallel 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 from 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 snt 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 excepted those specified by `unwantedlemmas` (if any).
  - `maxlemmas=n`: set the maximum number of selected axioms/lemmas to `n`. Keep this number small is generally more efficient. 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:

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

is equivalent to

```plaintext
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.
CHAPTER 3. TACTICS

58

admit

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

Type variables: <none>

x : int
y : int

\[ x = y + 1 \lor x = y - 1 \]

then running

\[ \text{admit}. \]

solves the goal.

change

Replace the current goal’s conclusion by \( \phi \) — \( \phi \) 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

\[ \text{true} \Rightarrow \text{true} \Leftrightarrow (y \Rightarrow \text{true} \lor z) \]

then running

\[ \text{change} \ (y \Rightarrow z). \]

produces the goal

Type variables: <none>

y : bool
z : bool
imp_yz: y => z

\[ y \Rightarrow z \]

pose \( x := \pi \)

Search for the first subterm \( t \) of the goal’s conclusion matching \( \pi \) 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 occurrence selector can be used (see Subsection 3.2.2). 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

\[ \text{pose} \ z := (_ + y) * _. \]
produces the goal

\[
\begin{align*}
\text{Type variables: } & \text{<none>} \\
x & : \text{int} \\
y & : \text{int} \\
z & : \text{int} := (x + 1 + y) \times x \\
2 \times z &= z + z
\end{align*}
\]

Logical cut. Generate two subgoals: one whose conclusion is the cut formula \(\phi\), and one with conclusion \(\phi \Rightarrow \psi\) where \(\psi\) is the current goal’s conclusion. Moreover, the introduction pattern \(\iota\) is applied to the second subgoal.

For example, if the current goal is

\[
\begin{align*}
\text{Type variables: } & \text{<none>} \\
x & : \text{bool} \\
\text{notnot}_x & : (x \Rightarrow \text{false}) \Rightarrow \text{false} \\
\end{align*}
\]

then running

\[
\text{cut excl_or : } x \lor (x \Rightarrow \text{false}).
\]

produces the goals

\[
\begin{align*}
\text{Type variables: } & \text{<none>} \\
x & : \text{bool} \\
\text{notnot}_x & : (x \Rightarrow \text{false}) \Rightarrow \text{false} \\
\end{align*}
\]

\[
\begin{align*}
x \lor (x \Rightarrow \text{false})
\end{align*}
\]

and

\[
\begin{align*}
\text{Type variables: } & \text{<none>} \\
x & : \text{bool} \\
\text{notnot}_x & : (x \Rightarrow \text{false}) \Rightarrow \text{false} \\
excl_or & : x \lor (x \Rightarrow \text{false})
\end{align*}
\]

\[
x
\]

Syntax: \texttt{have } \iota : \phi \texttt{ by } \tau. Attempts to use tactic \(\tau\) to close the first subgoal (corresponding to the cut formula \(\phi\)), and fails if impossible.

Same as \texttt{have} (p. 59).

Syntax: \texttt{apply } p \mid \texttt{apply } /p_1 \cdots /p_n \mid \texttt{apply } p \texttt{ in } \overline{H}

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
CHAPTER 3. TACTICS

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 · · · /p_n. Apply the proof terms p_1, ..., 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
CHAPTER 3. TACTICS

Type variables: <none>

\( x : \text{int} \)

then running

\[ \text{apply} \ R \ /Q \ /P \ /P. \]

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>

\( x : \text{int} \)

\( \text{HP: } P \ (x + 1) \)

\( Q \ x \Rightarrow R \ x \)

then running

\[ \text{apply} \ R \ \text{in} \ \text{HP}. \]

produces the goal

Type variables: <none>

\( x : \text{int} \)

\( \text{HP: } Q \ x \Rightarrow R \ x \)

\( Q \ x \Rightarrow R \ x \)

\[ \text{exact} \ p | \text{exact} \ /p_1 \ldots /p_n \]

**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 \( /p_1 \ldots /p_n \). Equivalent to by apply \( /p_1 \ldots /p_n \).

\[ \text{rewrite} \ \pi_1 \ldots \pi_n | \text{rewrite} \ \pi_1 \ldots \pi_n \ \text{in} \ H \]

**Syntax:** rewrite \( \pi_1 \ldots \pi_n \). Rewrite the rewrite-pattern \( \pi_1 \ldots \pi_n \) from left to right, where the \( \pi_i \) can be of the following form:

- one of \(/, /\pi, /\pi\),
- 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 3.2.2), followed by an occurrence selector (see Subsection 3.2.2), followed (for proof-terms only) by a repetition marker (\( !, ?, n! \) or \( n? \)). All these prefixes are optional. Depending on the form of \( \pi \), rewrite \( \pi \) does the following:

- For \(/, /\pi\), and \(/\pi\), see Subsection 3.2.3.
• If \( \pi \) is a proof-term with conclusion \( f_1 = f_2 \), then \texttt{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 assumptions 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 \( f_1 \Leftrightarrow f_2 \). If the direction indicator - is given, \texttt{rewrite} works in the reverse direction, searching for a match of \( f_2 \) and then replacing all occurrences of \( f_1 \) by \( f_2 \) in the goal’s conclusion, and creates new subgoals for the instantiations of the assumptions of \( p \).

• If \( \pi \) is a \:/-prefixed pattern of the form \( o p_1 \cdots p_n \), with \( o \) a defined symbol, then \texttt{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 \( \delta \)-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, \texttt{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 3.2.2.

Repetition markers allow the repetition of the same rewriting. For instance, \texttt{rewrite} \( \pi \) leads to \texttt{do \! rewrite} \( \pi \). See the tactical \texttt{do} for more information.

Lastly, \texttt{rewrite} \( \pi_1 \cdots \pi_n \) is equivalent to \texttt{rewrite} \( \pi_1 \); \ldots; \texttt{rewrite} \( \pi_n \).

For example, if the current goal is

```plaintext
Type variables: <none>

x : int
y : int
eq_xy: x = y

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

then running

```plaintext
rewrite eq_xy.
```

produces

```plaintext
Type variables: <none>

x : int
y : int
eq_xy: x = y

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

from which running

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

produces

```plaintext
Type variables: <none>

x : int
y : int
eq_xy: x = y

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

\[
\text{rewrite } \text{eq}_\text{xy}.
\]

produces

Type variables: <none>

\[
\begin{align*}
x & : \text{int} \\ y & : \text{int} \\ \text{eq}_\text{xy} & : x = y
\end{align*}
\]

\[
\text{forall } (z : \text{int}), x = z \Rightarrow x = z
\]

Syntax: \text{rewrite } \pi_1 \cdots \pi_n \text{ in } H. \text{ Like the preceding case, except rewriting is done in the hypothesis } H \text{ 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>

\[
\begin{align*}
x & : \text{int} \\ y & : \text{int} \\ \text{eq}_\text{xy} & : x = y \\ z & : \text{int} \\ \text{eq}_\text{xz} & : y = z
\end{align*}
\]

\[
x = z
\]

then running

\[
\text{rewrite } \text{eq}_\text{xy} \text{ in } \text{eq}_\text{xz}.
\]

produces

Type variables: <none>

\[
\begin{align*}
x & : \text{int} \\ y & : \text{int} \\ \text{eq}_\text{xy} & : x = y \\ z & : \text{int} \\ \text{eq}_\text{xz} & : x = z
\end{align*}
\]

\[
x = z
\]

\[
\text{case } \phi | \text{ case } φ.
\]

Syntax: \text{case } φ. \text{ Do an excluded-middle case analysis on } φ, \text{ substituting } φ \text{ in the goal’s conclusion. For example, if the current goal is}

Type variables: <none>

\[
\begin{align*}
x & : \text{int} \\ y & : \text{int} \\ \text{abs}_\text{bnd} & : \lvert x - y \rvert <= 10
\end{align*}
\]

\[
x - y <= 10 \lor y - x <= 10
\]

then running

\[
\text{case } (x <= y).
\]
produces the goals

\[
\text{Type variables: } \text{<none>}
\]
\[
\begin{aligned}
&x : \text{int} \\
y : \text{int} \\
&\text{abs\_bnd: } \|x - y\| \leq 10 \\
\end{aligned}
\]
\[
\begin{aligned}
x \leq y \Rightarrow x - y \leq 10 \lor y - x \leq 10 \\
\end{aligned}
\]

and

\[
\text{Type variables: } \text{<none>}
\]
\[
\begin{aligned}
x : \text{int} \\
y : \text{int} \\
&\text{abs\_bnd: } \|x - y\| \leq 10 \\
\end{aligned}
\]
\[
\begin{aligned}
! x \leq y \Rightarrow x - y \leq 10 \lor y - x \leq 10 \\
\end{aligned}
\]

**Syntax:** \texttt{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 3.2.3.*

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

\[
\text{Type variables: } \text{<none>}
\]
\[
\begin{aligned}
x : \text{int} \\
y : \text{int} \\
z : \text{int} \\
\end{aligned}
\]
\[
\begin{aligned}
x = y \land & y = z \Rightarrow x = z \\
\end{aligned}
\]
then running

\texttt{case.}

produces the goal

\[
\text{Type variables: } \text{<none>}
\]
\[
\begin{aligned}
x : \text{int} \\
y : \text{int} \\
z : \text{int} \\
\end{aligned}
\]
\[
\begin{aligned}
x = y \Rightarrow y = z \Rightarrow x = z \\
\end{aligned}
\]

\&\& works identically.

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

\[
\text{Type variables: } \text{<none>}
\]
\[
\begin{aligned}
x : \text{int} \\
y : \text{int} \\
\end{aligned}
\]
\[
\begin{aligned}
x < y \lor y < x \Rightarrow x - y \neq 0 \\
\end{aligned}
\]
then running

\texttt{case.}
produces the goals

\[
\begin{align*}
\text{Type variables: } & <\text{none}> \\
x : \text{int} \\
y : \text{int} \\
x < y \Rightarrow x - y \neq 0
\end{align*}
\]

and

\[
\begin{align*}
\text{Type variables: } & <\text{none}> \\
x : \text{int} \\
y : \text{int} \\
y < x \Rightarrow x - y \neq 0
\end{align*}
\]

\[\|\] works identically.

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

\[
\begin{align*}
\text{Type variables: } & <\text{none}> \\
\exists (x : \text{int}), 0 < x < 5 \Rightarrow \exists (x : \text{int}), 5 < x < 10
\end{align*}
\]

then running

\[
\text{case.}
\]

produces the goal

\[
\begin{align*}
\text{Type variables: } & <\text{none}> \\
\forall (x : \text{int}), 0 < x < 5 \Rightarrow \exists (x_0 : \text{int}), 5 < x_0 < 10
\end{align*}
\]

- **(unit)** Substitutes \(\text{tt}\) for the assumption in the remainder of the conclusion.

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

\[
\begin{align*}
\text{Type variables: } & <\text{none}> \\
\forall (x \ y : \text{bool}), x \iff y \Rightarrow x = y
\end{align*}
\]

then running

\[
\text{case.}
\]

produces the goals

\[
\begin{align*}
\text{Type variables: } & <\text{none}> \\
\forall (y : \text{bool}), \text{true} \iff y \Rightarrow \text{true} = y
\end{align*}
\]

and
• (product type) For example, if the current goal is

\[
\text{forall } (x, y : \text{bool} \times \text{bool}), \quad x \not= y \Rightarrow x.1 \not= y.1 \lor x.2 \not= y.2
\]

then running

\text{case}.

produces the goal

\[
\text{forall } (x_1, x_2 : \text{bool} \times \text{bool}), (y : \text{bool} \times \text{bool}), \quad (x_1, x_2) \not= y \Rightarrow (x_1, x_2).1 \not= y.1 \lor (x_1, x_2).2 \not= y.2
\]

• (inductive datatype) Consider the inductive datatype declaration:

\[
\text{type } \text{tree} = \text{Leaf} \mid \text{Node } \text{of bool} & \text{tree} & \text{tree}.
\]

Then, if the current goal is

\[
\text{forall } (tr : \text{tree}), \quad tr = \text{Leaf} \lor \exists (b : \text{bool}) (tr_1 \ tr_2 : \text{tree}), tr = \text{Node } b \ tr_1 \ tr_2
\]

then running

\text{case}.

produces the goals

\[
\text{Leaf} = \text{Leaf} \lor \exists (b : \text{bool}) (tr_1 \ tr_2 : \text{tree}), \text{Leaf} = b \ tr_1 \ tr_2
\]

and

\[
\text{forall } (b : \text{bool}) (t \ t_0 : \text{tree}), \quad \text{Node } b \ t \ t_0 = \text{Leaf} \lor \exists (b_0 : \text{bool}) (tr_1 \ tr_2 : \text{tree}), \quad \text{Node } b \ t \ t_0 = b_0 \ tr_1 \ tr_2
\]
**CHAPTER 3. TACTICS**

Elim

**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 3.2.3.*

Elim mostly works identically to *case* (p. 63), 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)
```

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)
```

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)
```

**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 3.2.3.* 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 \texttt{Int} theory in Chapter 5, we’ll discuss the induction principle on the integers.

\textbullet algebra

\textbf{FiXme} Note: Missing description of \texttt{algebra}.

\section*{3.2.5 Tacticals}

Tactics can be combined together, composed and modified by \textit{tacticals}. We’ve already seen the introduction and generalization tacticals, which turn a tactic $\tau$ and a list of patterns into a composite tactic, which may then combined with other tactics.

\textbullet $\tau_1; \tau_2$

Apply $\tau_2$ to all the subgoals generated by $\tau_1$. Sequencing groups to the left, so that $\tau_1; \tau_2; \tau_3$ means $(\tau_1; \tau_2); \tau_3$.

For example, if the current goal is

Type variables: <none>

x : bool
y : bool
z : bool

$(x \land y) \land z => x \land y \land z$

then running

\texttt{case; case.}

produces the goals
CHAPTER 3. TACTICS

69

Type variables: <none>
x : bool
y : bool
z : bool

\( x \Rightarrow y \Rightarrow z \Rightarrow x \land y \land z \)

\( \tau_1; [\tau_1 | \cdots | \tau_n] \)

Run \( \tau_1 \), which must generate exactly \( n \) subgoals, \( G_1, \ldots, G_n \). Then apply \( \tau'_i \) 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

\( x \land y \land z \)

then running

\( \text{split}; [\text{assumption} | \text{split}; \text{assumption}] \)

solves the goal.

\( \text{try } \tau \)

Execute the tactic \( \tau \) 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.

\( \text{do! } \tau \)

Apply \( \tau \) to the current goal, then repeatedly apply it to all subgoals, stopping on a branch only when it fails. An error is produced if \( \tau \) 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

\( ((x \land y) \land z) \land w \lor (x \land y \land z) \land w \Rightarrow w \)

then running

\( \text{do! case} \)

produces the goals
CHAPTER 3. TACTICS

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 \(\setminus\) z ⇒ w ⇒ w

Variants.

\(\text{do} \, \tau\) apply \(\tau\) 0 or more times, until it fails
\(\text{do } n! \, \tau\) apply \(\tau\) with depth exactly \(n\)
\(\text{do } n? \, \tau\) apply \(\tau\) with depth at most \(n\)

\(\tau; \, \text{first } \tau_2\)

Apply the tactic \(\tau_1\), then apply \(\tau_2\) on the first subgoal generated by \(\tau_1\), leaving the other goals unchanged. An error is produced if no subgoals are generated by \(\tau_1\).

Variants.

\(\tau_1; \, \text{first } n \, \tau_2\) apply \(\tau_2\) on the first \(n\) subgoals generated by \(\tau_1\)
\(\tau_1; \, \text{last } \tau_2\) apply \(\tau_2\) on the last subgoal generated by \(\tau_1\)
\(\tau_1; \, \text{last } n \, \tau_2\) apply \(\tau_2\) on the last \(n\) subgoals generated by \(\tau_1\)
\(\tau; \, \text{first } n \, \text{last}\) reorder the subgoals generated by \(\tau\), moving the first \(n\) to the end of the list
\(\tau; \, \text{last } n \, \text{first}\) reorder the subgoals generated by \(\tau\), moving the last \(n\) to the beginning of the list
\(\tau; \, \text{last } \text{first}\) reorder the subgoals generated by \(\tau\), moving the last one to the beginning of the list
\(\tau; \, \text{first } \text{last}\) reorder the subgoals generated by \(\tau\), moving the first one to the end of the list

For example, if the current goal is

\(\text{Type variables: <none>}

x : bool
y : bool
z : bool

(x \(\setminus\) y ⇒ z) ⇔ (x ⇒ y ⇒ z)

then running
produces the goals

Type variables: <none>

\[ (x \Rightarrow y \Rightarrow z) \Rightarrow x \land y \Rightarrow z \]

and

Type variables: <none>

\[ (x \lor y \Rightarrow z) \Rightarrow x \Rightarrow y \Rightarrow z \]

by \( \tau \)

Apply the tactic \( \tau \) 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

\[ \text{lemma} \ldots \text{by} [] \]

means

\[ \text{lemma} \ldots \text{by} (\text{trivial}; \text{smt}). \]

### 3.3 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, strengthening, 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 \( \tau\{1\} \) or \( \tau\{2\} \)), a one-sided variant is used, with 1 referring to the left program, and 2 to the right one.
3.3.1 Tactics for Reasoning about Programs

\(\text{proc}\)

**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 \(\text{res} / \text{res}^i\) are turned into ones about the value(s) returned by the procedure(s).

For example, if the current goal is

Type variables: <none>

\[\begin{align*}
\text{pre} &= \{x, y\} \\
G1.f &\sim G2.f \\
\text{post} &= \{\text{res}\}
\end{align*}\]

then running

\(\text{proc}.\)

produces the goal

Type variables: <none>

\[\begin{align*}
&1 (\text{left}) : G1.f \\
&2 (\text{right}) : G2.f \\
\text{pre} &= \\
(x(1), y(1)).'1 = (x(2), y(2)).'1 \land \\
(x(1), y(1)).'2 = (x(2), y(2)).'2 \\
x &\leftarrow x + 1 \\
y &\leftarrow y + 1 \\
\text{post} &= x(1) + y(1) = x(2) + y(2)
\end{align*}\]

**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 procedure) 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

\[
\begin{align*}
\text{module type OR = } \\
&\text{proc f1() : unit} \\
&\text{proc f2() : unit} \\
&\text{proc f3() : unit}
\end{align*}
\]
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
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

```pseudocode
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 = {
```
\begin{verbatim}
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
\end{verbatim}
CHAPTER 3. TACTICS

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)

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

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 \& 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 \nOr1.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} \n={glob Adv} \nOr1.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} \n={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} \nOr1.qry{1} = Or2.qry{2} \ Or1.queried{1} = Or2.queried{2}

and
CHAPTER 3. TACTICS

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.
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 $P \Rightarrow Q$, 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 n1 n2 : 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

\[
\text{seq } 1 \ 1 : (x{1} = x{2} + 1 \land y{2} = y{1} + 1).
\]

produces the goals

\[
\text{Type variables: <none>}
\]

\[
&1 \ \text{(left)} : G1.f \\
&2 \ \text{(right)} : G2.f
\]

\[
\text{pre = } = (x, y) \\
x \leftarrow x + 1 \quad (1) \ y \leftarrow y + 1 \\
\text{post = } x{1} = x{2} + 1 \land y{2} = y{1} + 1
\]

and

\[
\text{Type variables: <none>}
\]

\[
&1 \ \text{(left)} : G1.f \\
&2 \ \text{(right)} : G2.f
\]

\[
\text{pre = } x{1} = x{2} + 1 \land y{2} = y{1} + 1 \\
y \leftarrow y + 1 \quad (1) \ x \leftarrow x + 1 \\
\text{post = } x{1} + y{1} = x{2} + y{2}
\]

**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

\[
\text{Type variables: <none>}
\]

\[
\text{Context : M.f} \\
\text{pre = } x \mod 2 = 0 \\
(1) \ x \leftarrow x + 1 \\
(2) \ x \leftarrow x + 1 \\
\text{post = } x \mod 2 = 0
\]

then running

\[
\text{seq } 1 : (x \mod 2 = 1).
\]

produces the goals
CHAPTER 3. TACTICS

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

\section*{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>

\begin{verbatim}
&1 (left) : M.f
&2 (right) : N.f

pre = =\{y\}
if (y = 0) {
  y <- 2
} else {
  y <- y - 2
} x <@ M.g() 

post = x\{1\} + y\{1\} = x\{2\} + y\{2\} - 1
\end{verbatim}

then running

\begin{verbatim}
sp.
\end{verbatim}

produces the goal

Type variables: <none>
&1 (left) : M.f  
&2 (right) : N.f

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

post = x(1) + y(1) = x(2) + y(2) - 1

Syntax: \( sp \ n_1 \ n_2 \). In \( \text{prhl} \), let \( sp \) consume \textit{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 \( \text{phl} \) and \( \text{hl} \), let \( sp \) consume \textit{exactly} \( n \) statements of the program. Fails if this isn’t possible.

\( \text{wp} \)

Syntax: \( wp \). If the goal’s conclusion is a \( \text{prhl} \), \( \text{phl} \) or \( \text{hl} \) 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 \textit{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

(1--) x <$> [1..10]

if (y = 0) {
  y <- 2  
} else {
  y <- y - 2
}

post = x(1) + y(1) = x(2) + y(2) - 1

then running

wp.

produces the goal

Type variables: <none>
\[ \text{pre } = \{ y \} \]
\[ \text{post } = \begin{cases} 
\text{if } y_2 = 0 \text{ then} \\
\quad \begin{cases} 
\text{if } y_1 = 0 \text{ then } x_1 + 2 = x_2 + 3 - 1 \\
\quad \text{else } x_1 + (y_1 - 2) = x_2 + 3 - 1 
\end{cases} \\
\text{else } \begin{cases} 
\text{let } y_R = y_2 - 1 \text{ in} \\
\quad \begin{cases} 
\text{if } y_1 = 0 \text{ then } x_1 + 2 = x_2 + y_R - 1 \\
\quad \text{else } x_1 + (y_1 - 2) = x_2 + y_R - 1 
\end{cases} 
\end{cases} 
\end{cases} 
\]

**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.

\( \text{rnd} \)

When describing the variants of this tactic, we only 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:** \( \text{rnd} \mid \text{rnd} \ f \mid \text{rnd} \ f \ g \). 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

\[ \text{Type variables: <none>} \]
\[ n : \text{int} \]
\[ \&1 (\text{left}) : M.f \]
\[ \&2 (\text{right}) : N.f \]
\[ \text{pre } = y_2 = n \]
\[ \text{post } = x_1 \leftrightarrow x_2 + y_2 = n + 2 \]

then running
\texttt{rnd (fun b => b \ ? \ 3 : 2) (fun m => m = 3)}.

produces the goal

\begin{verbatim}
Type variables: <none>

n : int

&1 (left ) : M.h
&2 (right) : M.h

pre = y{2} = n

(1) y <- y - 1

post =
  (forall (xR : int),
   in_supp xR [2..3] => xR = if xR = 3 then 3 else 2) & &
  (forall (xR : int),
   in_supp xR [2..3] => mu_x [2..3] xR = mu_x {0,1} (xR = 3)) & &
  forall (xL : bool),
   in_supp xL {0,1} =>
   in_supp (if xL then 3 else 2) [2..3] & &
   xL = (if xL then 3 else 2) = 3) & &
   (xL <=> (if xL then 3 else 2) + y{2} = n + 2)
\end{verbatim}

Note that if one uses the other isomorphism between \{0,1\} and [2..3] the generated subgoal will be false.

\textbf{Syntax:} \texttt{rnd\{1\} | rnd\{2\}}. If the conclusion is a \textsc{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:

\begin{itemize}
  \item the weight of $d$ is 1 (so the random assignment can’t fail);
  \item 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.
\end{itemize}

For example, if the current goal is

\begin{verbatim}
Type variables: <none>

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

pre = true

x <$ \{0,1\}     (1) x <$ \{0,1\}

post = =x
\end{verbatim}

then running

\begin{verbatim}
 rnd\{1\}.
\end{verbatim}

produces the (false!) goal

\begin{verbatim}
Type variables: <none>
\end{verbatim}
`&1 (left) : M.f &2 (right) : M.f`  
`pre = true`  

(1)  \( x \in \{0,1\} \)

`post =`  

\( (weight \{0,1\})\%Distr = 1\%r \& \forall (x_0 : \text{bool}), (\text{in_supp} x_0 \{0,1\})\%Distr => x_0 = x{2} \)

**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)  \( y \leftarrow 2 \)
(2)  \( x \in \[1..10\] \)
post = \( 3 <= x + y <= 12 \)
```

then running  
```
rnd.
```

produces the goal  
```
Type variables: <none>
```

```
Context : M.f
pre = true
(1)  \( y \leftarrow 2 \)
post = \( \forall (x_0 : \text{int}), \text{in_supp} x_0 [1..10] => 3 <= x_0 + y <= 12 \)
```

**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 \texttt{(left) : M.f}
\&2 \texttt{(right) : N.f}

\texttt{pre = =(x, y)}

\texttt{if (x > y) \{ (1--) \texttt{if (y < x) \{}}
\texttt{z \leftarrow x - y} \quad \texttt{(1.1) z \leftarrow x - y + 1}
\texttt{\} else \{ (1--)} \texttt{else \{}}
\texttt{z \leftarrow y - x} \quad \texttt{(1?1) z \leftarrow y - x + 1}
\texttt{\} \texttt{else \{} \texttt{z \leftarrow z * 2} \quad \texttt{(2--)}}
\texttt{post = z\{1\} >= 0 /\ z\{1\} = 2 * (z\{2\} - 1)}
```

then running

```
if.
```

produces the goals

```
Type variables: <none>

\texttt{forall \&1 \&2, -(x, y) => x\{1\} > y\{1\} \Leftrightarrow y\{2\} < x\{2\}}
```

and

```
Type variables: <none>

\&1 \texttt{(left) : M.f}
\&2 \texttt{(right) : N.f}

\texttt{pre = =(x, y) /\ x\{1\} > y\{1\}}

\texttt{z \leftarrow x - y} \quad \texttt{(1) z \leftarrow x - y + 1}
\texttt{z \leftarrow z * 2} \quad \texttt{(2)}

\texttt{post = z\{1\} >= 0 /\ z\{1\} = 2 * (z\{2\} - 1)}
```

and

```
Type variables: <none>

\&1 \texttt{(left) : M.f}
\&2 \texttt{(right) : N.f}

\texttt{pre = =(x, y) /\ ! x\{1\} > y\{1\}}

\texttt{z \leftarrow y - x} \quad \texttt{(1) z \leftarrow y - x + 1}
\texttt{z \leftarrow z * 2} \quad \texttt{(2)}

\texttt{post = z\{1\} >= 0 /\ z\{1\} = 2 * (z\{2\} - 1)}
```
**Syntax:** \( \text{if} \{1\} \mid \text{if} \{2\} \). 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 (x > y) {
  z <- x - y
} else {
  z <- y - x
}
else {
  z <- z * 2
}
post = z{1} >= 0 /\ z{1} = 2 * (z{2} - 1)
```

then running

```
if\{i\}.
```

produces the goals

```
Type variables: <none>

\&1 (left) : M.f
\&2 (right) : N.f
pre = ={x, y} /\ x(1) > y(1)
if (x > y) {
  z <- x - y
} else {
  z <- y - x
}
else {
  z <- z * 2
}
post = z{1} >= 0 /\ z{1} = 2 * (z{2} - 1)
```

and

```
Type variables: <none>

\&1 (left) : M.f
\&2 (right) : N.f
pre = ={x, y} /\ ! x{1} > y{1}
```
CHAPTER 3. TACTICS

\[ z \leftarrow y - x \]
\[ \text{(1--)} \quad \text{if} \quad (y < x) \{ \]
\[ \text{(1.1)} \quad z \leftarrow x - y + 1 \]
\[ \text{(1--) } \quad \text{else} \{ \]
\[ \text{(1?1)} \quad z \leftarrow y - x + 1 \]
\[ \text{(1--) } \}
\[ \]
\[ z \leftarrow z * 2 \]
\[ \text{(2--)} \]
\[ \]
\[ \text{post} = z \{1\} \geq 0 \land z \{1\} = 2 * (z \{2\} - 1) \]

**Syntax:** 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--) \quad \text{if} \quad (x > y) \{ 
(1.1) \quad z \leftarrow x - y 
(1--) \} \quad \text{else} \{ 
(1?1) \quad z \leftarrow y - x 
(1--) \}
(2--) \quad z \leftarrow z * 2
post = z \geq 0

then running

*if.*

produces the goals

Type variables: <none>

Context : M.f

pre = x > y

(1) \quad z \leftarrow x - y
(2) \quad z \leftarrow z * 2
post = z \geq 0

and

Type variables: <none>

Context : M.f
\[ \text{pre} = \neg x > y \]

(1)  \[ z \leftarrow y - x \]
(2)  \[ z \leftarrow z * 2 \]

\[ \text{post} = z \geq 0 \]

\section*{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 \texttt{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 \texttt{while} statements, whose precondition is the conjunction of \( I \) and the \texttt{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 \texttt{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 \texttt{while} statements from the two programs, and whose postcondition is the conjunction of:
  - the conjunction of \( I \) and the assertion that the \texttt{while} statements’ boolean expressions are equivalent; and
  - the assertion that, for all values of the variables \textit{modified} by the \texttt{while} statements, if the \texttt{while} statements’ boolean expressions don’t hold, but \( I \) holds, then the original goal’s postcondition holds (in \( I \), the \texttt{while} statements’ boolean expressions, and the postcondition, variables modified by the \texttt{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

\begin{verbatim}
Type variables: <none>

n : int

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

pre = \{x, y\} \land y{1} = n

z <- 0
while (x > 0) {
    z <- z + 1
    x <- x - 1
}

\texttt{while} (i <= x) {
    \texttt{while} (i <= x) {
        z <- z + 2
        i <- i + 1
    }
}

post = (z{1} + y{1} - n) * 2 + 1 = z{2} + y{2} - n
\end{verbatim}
CHAPTER 3. TACTICS

```
while (x{1} - 1 = x{2} - i{2} \(\land\) z{1} * 2 + 1 = z{2}).
```

produces the goals

Type variables: <none>

\(n : \text{int}\)

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

\(\text{pre} = \)
\(\quad(x{1} - 1 = x{2} - i{2} \land z{1} * 2 + 1 = z{2}) \land\)
\(\quad x{1} > 0 \land i{2} <= x{2}\)
\(z \leftarrow z + 1\) \quad (1) \(z \leftarrow z + 2\)
\(x \leftarrow x - 1\) \quad (2) \(i \leftarrow i + 1\)

\(\text{post} = \)
\(\quad(x{1} - 1 = x{2} - i{2} \land z{1} * 2 + 1 = z{2}) \land\)
\(\quad (x{1} > 0 \iff i{2} <= x{2})\)

and

Type variables: <none>

\(n : \text{int}\)

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

\(\text{pre} = \{x, y\} \land y(1) = n\)
\(z \leftarrow 0\) \quad (1) \(z \leftarrow 1\)
\(\quad i \leftarrow i + 1\)

\(\text{post} = \)
\(\quad((x{1} - 1 = x{2} - i{2} \land z{1} * 2 + 1 = z{2}) \land\)
\(\quad (x{1} > 0 \iff i{2} <= x{2})) \land\)
\(\quad \forall (z_L \ x_L \ i_R \ z_R : \text{int}),\)
\(\quad \neg x_L > 0 \implies \)
\(\quad \neg i_R <= x{2} \implies\)
\(\quad x_L - 1 = x{2} - i_R \land z_L * 2 + 1 = z_R \implies\)
\(\quad (z_L + y(1) - n) * 2 + 1 = z_R + y(2) - n\)

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 non-designated program in `I` and `v` are interpreted in this memory; reference to the variables of the designated 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

\[
\begin{align*}
\&1 \text{ (left) : } M.f \\
\&2 \text{ (right) : } N.f \\
\text{pre} &= =\{n\} \land n1 \geq 0 \\
x &\leftarrow 0 \quad \text{(1--)} \\
i &\leftarrow 0 \quad \text{(2--)} \\
\text{while} \ (i < n) \{
\begin{align*}
x &\leftarrow x + (i + 1) \quad \text{(3.1)} \\
i &\leftarrow i + 1 \quad \text{(3.2)} \\
\}
\text{post} &= =\{x\} \leq n2 \times n2
\end{align*}
\end{align*}
\]

then running

\[
\text{while}\{1\} \ (0 \leq i1 \leq n1 \land x1 \leq i1 \times i1) \ (n1 - i1).
\]

produces the goals

\[
\begin{align*}
\text{forall } &\&m (z : \text{int}), \\
\text{phoare}\{ \begin{align*}
x &\leftarrow x + (i + 1); i &\leftarrow i + 1 : \\
(0 \leq i \leq n \land x \leq i \times i) \land i < n) \land n - i = z \implies \\
(0 \leq i \leq n \land x \leq i \times i) \land n - i < z \end{align*} = 1%r
\end{align*}
\]
and

\[
\begin{align*}
\&1 \text{ (left) : } M.f \\
\&2 \text{ (right) : } N.f \\
\text{pre} &= =\{n\} \land n1 \geq 0 \\
x &\leftarrow 0 \quad \text{(1)} \\
i &\leftarrow 0 \quad \text{(2)} \\
\text{post} = \\
(0 \leq i1 \leq n1 \land x1 \leq i1 \times i1) \land \\
\text{forall } (i_L x_L : \text{int}),
\end{align*}
\]
(0 <= i_L <= n{1} \ / x_L <= i_L * i_L =>
    \ n{1} - i_L <= 0 => 1 \ / (\ ! \ 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>

\(m : \text{int}\)

*Context:* \(M.f\)

\(pre = m = n \ \& \ n >= 0\)

(1--) \(x \leftarrow 0\)

(2--) \(i \leftarrow 0\)

(3--) \(\textbf{while} \ (i < n) \ {\}\)

(3.1) \(x \leftarrow x + (i + 1)\)

(3.2) \(i \leftarrow i + 1\)

(3--) \}

\(post = x <= m * m\)

then running

\(\textbf{while} \{1\} \ (0 <= i <= n \ \& \ x <= i * i).\)

produces the goals

*Type variables:* <none>

\(m : \text{int}\)

*Context:* \(M.f\)

\(pre = (0 <= i <= n \ \& \ x <= i * i) \ \& \ i < n\)

(1) \(x \leftarrow x + (i + 1)\)

(2) \(i \leftarrow i + 1\)

\(post = 0 <= i <= n \ \& \ x <= i * i\)
and

Type variables: <none>

\( m : \text{int} \)

Context: \( M.f \)

pre = \( m = n \lor n \geq 0 \)

\((1)\) \( x \leftarrow 0 \)

\((2)\) \( i \leftarrow 0 \)

post =

\( (0 \leq i \leq n \lor x \leq i \times i) \lor \)

\textbf{forall} \((i0 \ x0 : \text{int}),

\textbf{!} \ i0 < n \Rightarrow 0 \leq i0 \leq n \lor x0 \leq i0 \times i0 \Rightarrow x0 \leq m \times m \)

Syntax: \texttt{while I v. \text{PHL} version...}

\(\circ\)

call

When describing the variants of this tactic, we only 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: \texttt{call \(_: P \Rightarrow Q\).} If the goal’s conclusion is a \texttt{pRHL} statement judgement whose programs end with procedure calls or procedure call assignments (resp., an \texttt{HL} statement judgement whose program ends with a procedure call or procedure call assignment), then generate two subgoals:

- One whose conclusion is a \texttt{pRHL} judgement (resp., \texttt{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 \texttt{pRHL} statement judgement (resp., \texttt{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 = \( =\{u\}\)
and the procedures $M.f$ and $N.f$ have a single parameter, $y$, then running
\[
\text{call } (_: =\{y\} \land M.x{1} = -N.x{2} \Rightarrow =\{\text{res}\} \land M.x{1} = -N.x{2}).
\]
produces the goals

\[
\begin{align*}
\text{Type variables: } & \text{<none>} \\
\text{pre} = & \{y\} \land M.x{1} = -N.x{2} \\
M.f & - N.f \\
\text{post} = & =\{\text{res}\} \land M.x{1} = -N.x{2}
\end{align*}
\]

and

\[
\begin{align*}
\text{Type variables: } & \text{<none>} \\
\&1 & (\text{left} ) : M.g \\
\&2 & (\text{right} ) : N.g \\
\text{pre} = & =\{u\} \\
M.x & \leftarrow 5 \\
(1) & N.x \leftarrow -5 \\
\text{post} = & ( =\{u\} \land M.x{1} = -N.x{2} ) \&\& \\
\forall (\text{result}_L, \text{result}_R, x_L, x_R : \text{int}), \\
\text{result}_L = & \text{result}_R \land x_L = -x_R \Rightarrow \\
\text{result}_L + x_L = & \text{result}_R - x_R
\end{align*}
\]

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
\[
\text{lemma } M_N_f : \\
\equiv [M.f \sim N.f : \\
\{y\} \land M.x{1} = -N.x{2} \Rightarrow =\{\text{res}\} \land M.x{1} = -N.x{2}].
\]
then running
\[
\text{call } M_N_f.
\]
produces the goal

\[
\begin{align*}
\text{Type variables: } & \text{<none>} \\
\&1 & (\text{left} ) : M.g \\
\&2 & (\text{right} ) : N.g \\
\text{pre} = & =\{u\}
\end{align*}
\]
M.x <- 5 \quad (1) \quad N.x <- -5

post =
(\{u\} \land M.x = -N.x) \land \forall \text{result}_L \text{result}_R \text{x}_L \text{x}_R : \text{int},
\text{result}_L = \text{result}_R \land \text{x}_L = -\text{x}_R \Rightarrow
\text{result}_L + \text{x}_L = \text{result}_R - \text{x}_R

Syntax: \text{call}(1) (_ : P \Rightarrow Q) \mid \text{call}(2) (_ : P \Rightarrow 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

\[
\text{Type variables: <none>}
\]

\[
x2: \text{int}
\]

\[
\&1 (\text{left} ) : M.g \\
\&2 (\text{right} ) : N.g
\]

\[
\text{pre} = x2 = N.x = u = u \land u \equiv 2 = 2
\]

\[
\text{M.x} \leftarrow u \quad (1)
\]

\[
\text{M.f}(7) \quad (2)
\]

\[
\text{post} = (M.x \equiv 2 = 0) = (N.x \equiv 2 = 0)
\]

then running

\[
\text{call}(1) (_ : M.x \equiv 2 = x2 \equiv 2 \Rightarrow M.x \equiv 2 = x2 \equiv 2).
\]

produces the goals

\[
\text{Type variables: <none>}
\]

\[
x2: \text{int}
\]

\[
\text{pre} = M.x \equiv 2 = x2 \equiv 2
\]

\[
\text{M.f}
\]
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 \texttt{call}, in which case only the second subgoal need be generated.

For example, in the start-goal of the preceding example, if the lemma \texttt{M_f} is

\begin{verbatim}
lemma M_f (z : int) :
  phoare [M.f : M.x \equiv 2 = z \equiv 2 \Rightarrow M.x \equiv 2 = z \equiv 2] = 1%r.
\end{verbatim}

then running

\begin{verbatim}
call{1} (M_f x2).
\end{verbatim}

produces the goal

\begin{verbatim}
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} \equiv 2 = u{2} \equiv 2

M.x <- u

post = M.x{1} \equiv 2 = x2 \equiv 2 \&\&
    forall (x_L : int),
    x_L \equiv 2 = x2 \equiv 2 => (x_L \% 2 = 0) = (N.x{2} \% 2 = 0)
\end{verbatim}

\textbf{Syntax: \texttt{call} \(_ : I\).} If the conclusion is a \texttt{pRHL} statement judgement whose programs end with calls to \texttt{concrete} procedures (resp., an \texttt{HL} statement judgement whose program ends with a call to a concrete procedure), then use the specification argument to \texttt{call} generated from the \texttt{invariant} \_ and automatically apply \texttt{proc} to its first subgoal. In the \texttt{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

\texttt{call (} _ : M.x{1} = -N.x{2})\texttt{).}

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} \&
  \texttt{forall (result}_L \texttt{ result}_R \ _L \ _R : \texttt{int}),
result_L = result_R \land x_L = -x_R \Rightarrow
\quad 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} \land Or.x{1} \%\% 2 = 0 \land Or.x{2} \%\% 2 = 0

Adv(Or).g() (1) Adv(Or).g()

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

then running

```
call (_ : Or.x{1} \%\% 2 = 0 \land Or.x{2} \%\% 2 = 0).
```

produces the goals

```
Type variables: <none>
```
CHAPTER 3. TACTICS

Adv: $T\{Or\}$

\[
\begin{align*}
\text{pre} &= \text{true} \land \text{Or.x\{1\} } \% \% 2 = 0 \land \text{Or.x\{2\} } \% \% 2 = 0 \\
\text{Or.f1} &= \text{Or.f1} \\
\text{post} &= \{\text{res}\} \land \text{Or.x\{1\} } \% \% 2 = 0 \land \text{Or.x\{2\} } \% \% 2 = 0
\end{align*}
\]

and

Type variables: <none>

Adv: $T\{Or\}$

\[
\begin{align*}
\text{pre} &= \text{true} \land \text{Or.x\{1\} } \% \% 2 = 0 \land \text{Or.x\{2\} } \% \% 2 = 0 \\
\text{Or.f2} &= \text{Or.f2} \\
\text{post} &= \{\text{res}\} \land \text{Or.x\{1\} } \% \% 2 = 0 \land \text{Or.x\{2\} } \% \% 2 = 0
\end{align*}
\]

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 $\text{proc\ }B\ I\ \text{to its first subgoal}$, 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;
  }
```

### CHAPTER 3. TACTICS

#### 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

```latex
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) : M(Adv).h 

pre = Or1.qry{1} = Or2.qry{2}

\[
\begin{align*}
    b &\ll Adv(Or1).f() \quad (1) 
    \quad b \ll Adv(Or2).f()
\end{align*}
\]

post = !Or2.queried{2} => ={b} 

then running

```latex
call (\_ : Or2.queried, (Or1.qry{1} = Or2.qry{2})).
```

produces the goals

```latex
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} \(\land\) ={x} \(\land\) Or1.qry{1} = Or2.qry{2}

\[
\begin{align*}
    Or1.qry &\ll Or2.qry \\
    post &\ll !Or2.queried{2} \Rightarrow ={res} \land Or1.qry{1} = Or2.qry{2}
\end{align*}
\]

and

```latex
Type variables: <none>
```
CHAPTER 3. TACTICS

101

\[
\text{Adv: } \text{ADV}\{\text{Or1, Or2}\}
\]

\[
\text{l1}\_\text{Adv}\_f: \text{forall } (O <: \text{OR}\{\text{Adv}\}),
\]

\[
\text{islossless } O.\text{qry} \Rightarrow \text{islossless } \text{Adv}(O).f
\]

\[
\text{forall } &2, \text{Or2.queried}(2) \Rightarrow \text{islossless } \text{Or1.qry}
\]

and

Type variables: <none>

\[
\text{Adv: } \text{ADV}\{\text{Or1, Or2}\}
\]

\[
\text{l1}\_\text{Adv}\_f: \text{forall } (O <: \text{OR}\{\text{Adv}\}),
\]

\[
\text{islossless } O.\text{qry} \Rightarrow \text{islossless } \text{Adv}(O).f
\]

\[
\text{forall } &1,
\]

\[
\text{phoare}\{ \text{Or2.qry} : \text{Or2.queried} \land \text{true} \Rightarrow \text{Or2.queried} \land \text{true} \} = 1\%
\]

and

Type variables: <none>

\[
\text{Adv: } \text{ADV}\{\text{Or1, Or2}\}
\]

\[
\text{l1}\_\text{Adv}\_f: \text{forall } (O <: \text{OR}\{\text{Adv}\}),
\]

\[
\text{islossless } O.\text{qry} \Rightarrow \text{islossless } \text{Adv}(O).f
\]

\[
\&1 \text{ (left) } : \text{M}(\text{Adv}).h
\]

\[
\&2 \text{ (right) } : \text{N}(\text{Adv}).h
\]

\[
\text{pre} = \text{Or1.qry}(1) = \text{Or2.qry}(2)
\]

\[
\text{post} =
\]

\[
(\lnot \text{Or2.queried}(2) \Rightarrow \text{true} \land \text{Or1.qry}(1) = \text{Or2.qry}(2)) \land
\]

\[
\text{forall } (\text{result}_L, \text{result}_R : \text{bool}) (\text{Adv}_L, \text{Adv}_R : \text{glob Adv})
\]

\[
(\lnot \text{queried}_R \Rightarrow
\]

\[
\text{result}_L = \text{result}_R \land
\]

\[
\text{Adv}_L = \text{Adv}_R \land \text{Or1.qry}(1) = \text{Or2.qry}(2) \Rightarrow
\]

\[
\text{result}_R \Rightarrow \text{result}_L = \text{result}_R
\]

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>

\[
\text{Adv: } \text{ADV}\{\text{Or1, Or2}\}
\]
CHAPTER 3. TACTICS

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

&1 (left): K(Adv).h
&2 (right): N(Adv).h

pre = Or1.qry{1} = Or2.qry{2} \ Or1.queried{1} = Or2.queried{2}

b <@ Adv(Or1).f() \ 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

  &forall; (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

pre =
  !Or2.queried{2} \ ={x} \ 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
    ={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
3.3.2 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.

```plaintext
for all $k1, phoare [ Or2.qry :
  Or2.queried \ Or1.queried(1) = Or2.queried ==>
  Or2.queried \ Or1.queried(1) = Or2.queried ] = 1%r
```

Type variables: <none>

Adv: ADV(Or1, Or2)

11_Adv.f: for all ($O <: OR(Adv)),
  islossless $O.qry => islossless Adv(O).f

$k1$ (left $): M(Adv).h
$k2$ (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))
    (query_L : bool) (Adv_R : (glob Adv)) (query_R : bool),
    if query_R then query_L = query_R
    else
      result_L = result_R \ Adv_L = Adv_R \ Or1.qry(1) = Or2.qry(2) \ query_L = query_R =>
      query_L = query_R \ ((query_R => result_L = result_R)
```
Syntax: \texttt{swap} n k. Equivalent to \texttt{swap} \([n..n] k\).

Syntax: \texttt{swap} k. If \(k\) is non-negative, equivalent to \texttt{swap} 1 \(k\). If \(k\) is negative, equivalent to \texttt{swap} \(n\ k\), where \(n\) is the length of the program.

For example, suppose the current goal is

Type variables: \(<\text{none}>\)

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

\pre = \{M.x, M.y, M.z, M.w\}

\begin{align*}
M.x &< \neg \text{true} & (1) & M.x &< \text{true} \\
M.y &< \neg \text{false} & (2) & M.y &< \text{false} \\
M.z &< \neg \text{true} & (3) & M.z &< \text{true} \\
M.w &< \neg \text{false} & (4) & M.w &< \text{false}
\end{align*}

\post = \{M.x, M.y, M.z, M.w\}

Then running

\texttt{swap} 1 3 4.

produces goal

Type variables: \(<\text{none}>\)

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

\pre = \{M.x, M.y, M.z, M.w\}

\begin{align*}
M.z &< \neg \text{true} & (1) & M.z &< \text{true} \\
M.w &< \neg \text{false} & (2) & M.w &< \text{false} \\
M.x &< \neg \text{true} & (3) & M.x &< \text{true} \\
M.y &< \neg \text{false} & (4) & M.y &< \text{false}
\end{align*}

\post = \{M.x, M.y, M.z, M.w\}

From which running

\texttt{swap} 2 2.

produces goal

Type variables: \(<\text{none}>\)

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

\pre = \{M.x, M.y, M.z, M.w\}

\begin{align*}
M.z &< \neg \text{true} & (1) & M.z &< \text{true} \\
M.x &< \neg \text{true} & (2) & M.x &< \text{true} \\
M.y &< \neg \text{false} & (3) & M.y &< \text{false} \\
M.w &< \neg \text{false} & (4) & M.w &< \text{false}
\end{align*}

\post = \{M.x, M.y, M.z, M.w\}
From which running
\[
\text{swap}(1) \ [3 \ .. \ 4] -1.
\]
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 \quad (1) \ M.z ← true
M.y ← false \quad (2) \ M.x ← true
M.w ← false \quad (3) \ M.y ← false
M.x ← true \quad (4) \ M.w ← false

post = \{M.x, M.y, M.z, M.w\}

From which running
\[
\text{swap} 2.
\]
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 \quad (1) \ M.x ← true
M.w ← false \quad (2) \ M.y ← false
M.z ← true \quad (3) \ M.z ← true
M.x ← true \quad (4) \ M.w ← false

post = \{M.x, M.y, M.z, M.w\}

From which running
\[
\text{swap}(2) -1.
\]
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 \quad (1) \ M.x ← true
M.w ← false \quad (2) \ M.y ← false
M.z ← true \quad (3) \ M.w ← false
M.x ← true \quad (4) \ M.z ← true

post = \{M.x, M.y, M.z, M.w\}
**inline**

**Syntax:** `inline M_1.p_1 · · · 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 · · · M_n.p_n | inline(2) M_1.p_1 · · · 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 · · · 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

```plaintext
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 &2 (right) : M.h
pre = ={x}
b <@ M.g(x + 1)       (1) b <@ M.g(x + 1)
post = (!b{1}) = !b{2}
```

then running

```
inline M.g.
```

produces the goal

```
```
CHAPTER 3. TACTICS

Type variables: <none>

\[ \begin{align*}
\&1 \text{(left)} : & M.h \\
\&2 \text{(right)} : & M.h \\
\pre = & \{x\} \\
x_0 & \leftarrow x + 1 \\
M.y & <\emptyset M.f(x_0 - 1) \\
b & \leftarrow x_0 + M.y + 1 = 3 \\
\post = & (\neg b(1)) = \neg b(2) \\
\end{align*} \]

From which running \texttt{inline\{2\} M.f.}

produces the goal

Type variables: <none>

\[ \begin{align*}
\&1 \text{(left)} : & M.h \\
\&2 \text{(right)} : & M.h \\
\pre = & \{x\} \\
x_0 & \leftarrow x + 1 \\
M.y & <\emptyset M.f(x_0 - 1) \\
b & \leftarrow x_0 + M.y + 1 = 3 \\
x_1 & \leftarrow x_0 - 1 \\
x_1 & \leftarrow x_1 + 1 \\
M.y & \leftarrow x_1 * 2 \\
b & \leftarrow x_0 + M.y + 1 = 3 \\
\post = & (\neg b(1)) = \neg b(2) \\
\end{align*} \]

From which running \texttt{inline M.f.}

produces the goal

Type variables: <none>

\[ \begin{align*}
\&1 \text{(left)} : & M.h \\
\&2 \text{(right)} : & M.h \\
\pre = & \{x\} \\
x_0 & \leftarrow x + 1 \\
x_1 & \leftarrow x_0 - 1 \\
x_1 & \leftarrow x_1 + 1 \\
M.y & \leftarrow x_1 * 2 \\
b & \leftarrow x_0 + M.y + 1 = 3 \\
\post = & (\neg b(1)) = \neg b(2) \\
\end{align*} \]

And, if the current goal is

Type variables: <none>
CHAPTER 3. TACTICS

&1 (left) : M.h
&2 (right) : M.h

pre = =x

b <@ M.g(x + 1)          (1) b <@ M.g(x + 1)

post = (!b{1}) = !b{2}

then running

  inline*.

produces the goal

Type variables: <none>

&1 (left) : M.h
&2 (right) : M.h

pre = =x

x0 <- x + 1          (1) x0 <- x + 1
x1 <- x0 - 1          (2) x1 <- x0 - 1
x1 <- x1 + 1          (3) x1 <- x1 + 1
M.y <- x1 * 2         (4) M.y <- x1 * 2
b <- x0 + M.y + 1 = 3 (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 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 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 $\in\{0,1\}$
(2--) y <- x ^* x
(3--) if (!y) {
(3.1) z <- true
(3--) } else {
(3?1) z <- false
(3--) }

post = z
then running

\texttt{rcondt 3.}

produces the goals

Type variables: <none>

Context : M.f

pre = true

(1) x < $\{0,1\}$
(2) y $\leftarrow x \downarrow x$

post = !y

and

Type variables: <none>

Context : M.f

pre = true

(1) x < $\{0,1\}$
(2) y $\leftarrow x \downarrow x$
(3) z $\leftarrow$ true

post = z

Syntax: $\texttt{rcondt}\{1\} n \mid \texttt{rcondt}\{2\} n$. If the goal’s conclusion is a pRHL statement judgement
where the $n$th statement of the designated program is an $\texttt{if}$ 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 $\texttt{if}$ 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 $\texttt{if}$ statement has been replaced by its $\texttt{then}$ part.

For example, if the current goal is

Type variables: <none>

\texttt{&1 (left) : N.f}
\texttt{&2 (right) : M.f}

pre = N.x{1}

(1--) x < $\{0,1\}$
(2--) y $\leftarrow x \downarrow x$
(3--) \texttt{if (!y) { (3.1) z $\leftarrow$ true} }
CHAPTER 3. TACTICS

(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}

© rcondf

Syntax: rcondf n. If the goal’s conclusion is an HL statement judgement whose n-th statement 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 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

\texttt{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: \texttt{rcondf\{1\} n | rcondf\{2\} n}. If the goal’s conclusion is a \texttt{pRHL} statement judgement where the \textit{n}th statement of the designated program is an \texttt{if} statement, reduce the goal to two subgoals.

- One whose conclusion is an \texttt{HL} statement judgement whose precondition is the original goal’s precondition, program is the first \textit{n} − 1 statements of the original goal’s designated program, and postcondition is the negation of the boolean expression of the \texttt{if} statement. Actually, the \texttt{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 \texttt{pRHL} statement judgement that’s the same as that of the original goal except that the \texttt{if} statement has been replaced by its \texttt{else} part.

For example, if the current goal is

Type variables: <none>

\texttt{&1 (left) : N.f}  
\texttt{&2 (right) : M.f}
pre = !N.x{1}  
(1--) x <\$ \{0,1\} 
(2--) y <- x "" x  
(3--) \texttt{if (y) \{  
(3.1) z <- true}
\[\begin{align*}
\text{(3--) } & \text{ else } \\
\text{(3?1) } & \quad z \leftarrow \text{ false} \\
\text{(3--) } & \end{align*}\]

\[\text{post} = N.x(1) = z(2)\]

then running

\[\text{rcondf(2) 3.}\]

produces the goals

Type variables: <none>

\[\text{forall } \&m, \text{ hoare}\left[ x <$ (0,1); y \leftarrow x ^ \wedge x : !N.x(\&m) \Rightarrow !y\right]\]

and

Type variables: <none>

\[\&1 \ (\text{left } ) : N.f \]
\[\&2 \ (\text{right } ) : M.f \]

\[\text{pre} = !N.x(1)\]

\[\begin{align*}
(1) & \quad x <$ (0,1) \\
(2) & \quad y \leftarrow x ^ \wedge x \\
(3) & \quad z \leftarrow \text{ false} \\
\text{post} & = N.x(1) = z(2) \\
\end{align*}\]

\textit{unroll}

\textbf{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>

\[\text{Context : M.f}\]

\[\text{pre} = \text{ true}\]

\[\begin{align*}
(1--) & \quad x \leftarrow 0 \\
(2--) & \quad z \leftarrow 0 \\
(3--) & \quad \text{while } (x < y) \{ \\
(3.1) & \quad z \leftarrow z + x \\
(3.2) & \quad x \leftarrow x + 1 \\
(3--) & \} \\
\text{post} & = z \geq 0 \\
\end{align*}\]

then running

\[\text{unroll 3.}\]

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 = z >= 0

And, if the current goal is

Type variables: <none>

Context : M.f

pre = true

(1----) if (y >= 0) {
    (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 = y >= 0

then running

unroll 1.2.

produces the goal

Type variables: <none>

Context : M.f

pre = true

(1----) if (y >= 0) {
    (1.1--) x <- 0
    (1.2--)
    (1.3--) while (x <> y) {
        (1.3.1) x <- x + 1
        (1.3--)
    (1----)
}
post = y >= 0

**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
&2 (right) : M.f

pre = ={y}

x <- 0
z <- 0
while (x < y) {
    z <- z + x
    x <- x + 1
}
```

```
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
z <- 0
if (x < y) {
    z <- z + x
    x <- x + 1
}
while (x < y) {
    z <- z + x
    x <- x + 1
}

post = ={z}
```

from which running

```
unroll(2) 3.
```

produces the goal

```
Type variables: <none>
```
\&1 (left) : M.f
\&2 (right) : M.f

\texttt{pre} = \{y\}
\begin{align*}
x & \leftarrow 0 \quad (1--) \quad x \leftarrow 0 \\
z & \leftarrow 0 \quad (2--) \quad z \leftarrow 0 \\
\textbf{if} (x < y) \{ \quad (3--) \quad \textbf{if} (x < y) \{ \\
  z & \leftarrow z + x \quad (3.1) \quad z \leftarrow z + x \\
x & \leftarrow x + 1 \quad (3.2) \quad x \leftarrow x + 1 \\
\} \quad (3--) \\
\textbf{while} (x < y) \{ \quad (4--) \quad \textbf{while} (x < y) \{ \\
  z & \leftarrow z + x \quad (4.1) \quad z \leftarrow z + x \\
x & \leftarrow x + 1 \quad (4.2) \quad x \leftarrow x + 1 \\
\} \quad (4--) \\
\end{align*}

\texttt{post} = \{z\}

\texttt{splitwhile}

**Syntax:** \texttt{splitwhile} \(c : e\). If the goal’s conclusion is an HL statement judgement whose \(c\)th statement is a \texttt{while} statement and \(e\) is a well-typed boolean expression in the \texttt{while} statement’s context, then insert before the \texttt{while} statement a copy of the \texttt{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 : \(M.f\)

\texttt{pre} = true
\begin{align*}
(1--) \quad x & \leftarrow 0 \\
(2--) \quad z & \leftarrow 0 \\
(3--) \quad \textbf{while} (x < y) \{ \\
(3.1) \quad z & \leftarrow z + x \\
(3.2) \quad x & \leftarrow x + 1 \\
(3--) \quad \}
\end{align*}

\texttt{post} = z \geq 0

then running

\texttt{splitwhile} \(3 : z \leq 20\).

produces the goal

Type variables: <none>

Context : \(M.f\)

\texttt{pre} = true
\begin{align*}
(1--) \quad x & \leftarrow 0 \\
(2--) \quad z & \leftarrow 0 \\
(3--) \quad \textbf{while} (x < y \land z \leq 20) \{ \\
(3.1) \quad z & \leftarrow z + x \\
(3.2) \quad x & \leftarrow x + 1 \\
\end{align*}
CHAPTER 3. TACTICS

(3--) }
(4--) while (x < y) {
(4.1) z <- z + x
(4.2) x <- x + 1
(4--) }

post = z >= 0

Syntax: \texttt{splitwhile}\{c : e | splitwhile\}{2} c : e. If the goal’s conclusion is a pRHL statement judgement where the \(c\)th statement of the designated program is a \texttt{while} statement and \(e\) is a well-typed boolean expression in the \texttt{while} statement’s context, then insert before the \texttt{while} statement a copy of the \texttt{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>

\[
\begin{align*}
&1 \textbf{(left )} : M.f \\
&2 \textbf{(right)} : M.f \\
\text{pre} &= \{y\} \\
& x <- 0 \quad (1--) \ x <- 0 \quad (2--) \ z <- 0 \quad (3--) \ \textbf{while} (x < y) \{ \quad (3.1) \ z <- z + x \quad (3.2) \ x <- x + 1 \quad (3--) \} \\
\text{post} &= \{z\} \\
\end{align*}
\]

then running

**splitwhile\{2\} 3 : z <= 20.**

produces the goal

Type variables: <none>

\[
\begin{align*}
&1 \textbf{(left )} : M.f \\
&2 \textbf{(right)} : M.f \\
\text{pre} &= \{y\} \\
& x <- 0 \quad (1--) \ x <- 0 \quad (2--) \ z <- 0 \quad (3--) \ \textbf{while} (x < y /\ z <= 20) \{ \quad (3.1) \ z <- z + x \quad (3.2) \ x <- x + 1 \quad (3--) \} \\
& \textbf{while} (x < y) \{ \quad (3--) \} \\
\text{post} &= \{z\} \\
\end{align*}
\]

from which running
 CHAPTER 3. TACTICS

\texttt{splitwhile(1) 3 : z <= 20.}

produces the goal

Type variables: <none>

\texttt{a1 (left) : M.f}
\texttt{a2 (right) : M.f}

\begin{align*}
\text{pre} & = =\{y\} \\
x \leftarrow 0 & \quad (1--) \quad x \leftarrow 0 \\
z \leftarrow 0 & \quad (2--) \quad z \leftarrow 0 \\
\text{while (x < y / z <= 20) \{ (3--) \quad \text{while (x < y / z <= 20) \{}} \\
z \leftarrow z + x & \quad (3.1) \quad z \leftarrow z + x \\
x \leftarrow x + 1 & \quad (3.2) \quad x \leftarrow x + 1 \\
\} & \quad (3--) \} \\
\text{while (x < y) \{ (4--) \quad \text{while (x < y) \{}} \\
z \leftarrow z + x & \quad (4.1) \quad z \leftarrow z + x \\
x \leftarrow x + 1 & \quad (4.2) \quad x \leftarrow x + 1 \\
\} & \quad (4--) \} \\
\text{post} & = =\{z\}
\end{align*}

\textbf{Fission}

\textbf{Syntax: fission} \texttt{c!l \& m, n.} HL statement judgement version. Fails unless \(0 \leq l\) and \(0 \leq m \leq n\) and the \(c\)th statement of the program is a \texttt{while} statement, and there are at least \(l\) statements right before the \texttt{while} statement, at its level, and the body of the \texttt{while} statement has at least \(n\) statements.

Let

- \(s_1\) be the \(l\) statements before the \texttt{while} statement at position \(c\);
- \(e\) be the boolean expression of the \texttt{while} statement;
- \(s_2\) be the first \(m\) statements of the body of the \texttt{while} statement;
- \(s_3\) be the next \(n - m\) statements of the body of the \texttt{while} statement;
- \(s_4\) be the rest of the body of the \texttt{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

\begin{align*}
\texttt{s_1 while (e) \{ s_2 s_3 s_4 \}}
\end{align*}

by

\begin{align*}
\texttt{s_1 while (e) \{ s_2 s_4 \}} \\
\texttt{s_1 while (e) \{ s_3 s_4 \}}
\end{align*}
For example, if the current goal is

Type variables: <none>

Context : M.f

\[pre = n > 0\]

(1--) \(x \leftarrow 0\)
(2--) \(y \leftarrow 0\)
(3--) \(i \leftarrow 0\)
(4--) \(j \leftarrow 0\)
(5--) \(while (i + j < n) \{
(5.1) \quad x \leftarrow x \ast i
(5.2) \quad x \leftarrow x + (j + 1)
(5.3) \quad y \leftarrow y \ast j
(5.4) \quad y \leftarrow y + (i + 2)
(5.5) \quad i \leftarrow i + 1
(5.6) \quad j \leftarrow j + 2
(5--) \}\)

\[post = x + y > 0\]

then running

\[fission 5!2 @ 2, 4.\]

produces the goal

Type variables: <none>

Context : M.f

\[pre = n > 0\]

(1--) \(x \leftarrow 0\)
(2--) \(y \leftarrow 0\)
(3--) \(i \leftarrow 0\)
(4--) \(j \leftarrow 0\)
(5--) \(while (i + j < n) \{
(5.1) \quad x \leftarrow x \ast i
(5.2) \quad x \leftarrow x + (j + 1)
(5.3) \quad i \leftarrow i + 1
(5.4) \quad j \leftarrow j + 2
(5--) \}\)

(6--) \(i \leftarrow 0\)
(7--) \(j \leftarrow 0\)
(8--) \(while (i + j < n) \{
(8.1) \quad y \leftarrow y \ast j
(8.2) \quad y \leftarrow y + (i + 2)
(8.3) \quad i \leftarrow i + 1
(8.4) \quad j \leftarrow j + 2
(8--) \}\)

\[post = x + y > 0\]

Syntax: \texttt{fission c} \& m, n. Equivalent to \texttt{fission c!1} \& m, n.

Syntax: \texttt{fission(1)} \cdots \mid \texttt{fission(2)} \cdots. The \texttt{PRHL} versions of the above variants, working on the designated program.
CHAPTER 3. TACTICS

fusion

Syntax: fusion c!l @m, n. HL statement judgement version. Fails unless 0 ≤ l and 0 ≤ m and 0 ≤ 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

\[
s_1 \text{ while } (e) \{ s_2 \ s_4 \}
\]

\[
s_1 \text{ while } (e) \{ s_3 \ s_4 \}
\]

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

\[
s_1 \text{ while } (e) \{ s_2 \ s_4 \}
\]

\[
s_1 \text{ while } (e) \{ s_3 \ s_4 \}
\]

by

\[
s_1 \text{ while } (e) \{ s_2 \ s_3 \ s_4 \}
\]

For example, if the current goal is

Type variables: <none>

Context : M.f

pre = n > 0

(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 = x + y > 0
then running

\[
\text{fusion} \ 5!2 @ 2, 2.
\]

produces the goal

Type variables: <none>

Context : M.f

\[
\text{pre} = n > 0
\]

(1--) \(x \leftarrow 0\)

(2--) \(y \leftarrow 0\)

(3--) \(i \leftarrow 0\)

(4--) \(j \leftarrow 0\)

(5--) \textbf{while} \ (i + j < n) \{

(5.1) \(x \leftarrow x * i\)

(5.2) \(x \leftarrow x + (j + 1)\)

(5.3) \(y \leftarrow y * j\)

(5.4) \(y \leftarrow y + (i + 2)\)

(5.5) \(i \leftarrow i + 1\)

(5.6) \(j \leftarrow j + 2\)

(5--) \}

\[
\text{post} = x + y > 0
\]

\textbf{Syntax:} fusion \(c \ 0 \ m, n\). Equivalent to fusion \(c!1 \ 0 \ m, n\).

\textbf{Syntax:} fusion\{1\} \ldots | fusion\{2\} \ldots. The pRHL versions of the above variants, working on the designated program.

\textbf{alias}

\textbf{Syntax:} alias \(c\) with \(x\). If the goal’s conclusion is an HL statement judgement whose program’s \(c\)th 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 \(x\).

\textbf{Syntax:} alias \(c\). Equivalent to alias \(c\) with \(x\).

\textbf{Syntax:} alias \(c\ . x = e\). If the program has an \(c\)th statement, and the expression \(e\) is well-typed in the context of the program, insert before the \(c\)th 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 \(x\).

\textbf{Syntax:} alias\{1\} \ldots | alias\{2\} \ldots. 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) \( x \leftarrow 10 \)
(2) \( (y, z) \leftarrow (x + 1, 6) \)

post = \( y + z = 17 \)

then running

\texttt{alias 2 with w.}

produces the goal

Type variables: <none>

Context : M.f

pre = true

(1) \( x \leftarrow 10 \)
(2) \( w \leftarrow (x + 1, 6) \)
(3) \( (y, z) \leftarrow w \)

post = \( y + z = 17 \)

from which running

\texttt{alias 3 u = w\textasciitilde1 + 7.}

produces the goal

Type variables: <none>

Context : M.f

pre = true

(1) \( x \leftarrow 10 \)
(2) \( w \leftarrow (x + 1, 6) \)
(3) \( u \leftarrow w\textasciitilde1 + 7 \)
(4) \( (y, z) \leftarrow w \)

post = \( y + z = 17 \)

from which running

\texttt{alias 3.}

produces the goal

Type variables: <none>

Context : M.f

pre = true

(1) \( x \leftarrow 10 \)
(2) \( w \leftarrow (x + 1, 6) \)
(3) \( x_0 \leftarrow w \cdot 1 + 7 \)
(4) \( u \leftarrow x_0 \)
(5) \( (y, z) \leftarrow w \)

post = y + z = 17

\begin{center}
\texttt{cfold}
\end{center}

\textbf{Syntax:} \texttt{c} \( m \). Fails unless \( n \geq 1 \) and \( m \geq 0 \). 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

\begin{center}
Type variables: <none>
\end{center}

\begin{center}
Context : M.f
\end{center}

\begin{center}
pre = true
\end{center}

\begin{center}
(1) \( x \leftarrow 1 \)
(2) \( y \leftarrow x + 1 \)
(3) \( z \leftarrow y + x + 2 \)
(4) \( w \leftarrow x - z \)
\end{center}

\begin{center}
post = w = -4
\end{center}

then running

\begin{center}
\texttt{c} \( 1 \) \( 1 \).
\end{center}

produces the goal

\begin{center}
Type variables: <none>
\end{center}

\begin{center}
Context : M.f
\end{center}

\begin{center}
pre = true
\end{center}

\begin{center}
(1) \( y \leftarrow 1 + 1 \)
(2) \( x \leftarrow 1 \)
(3) \( z \leftarrow y + x + 2 \)
(4) \( w \leftarrow x - z \)
\end{center}

\begin{center}
post = w = -4
\end{center}

from which running

\begin{center}
\texttt{c} \( 2 \).
\end{center}

produces the goal

\begin{center}
Type variables: <none>
\end{center}

\begin{center}
Context : M.f
\end{center}
pre = true

(1) y <- 1 + 1
(2) z <- y + 1 + 2
(3) w <- 1 - z
(4) x <- 1

post = w = -4

from which running

\texttt{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

\texttt{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

\textbf{Syntax:} \texttt{cfold\{1\} c \ | \ cfold\{2\} c \ \dagger \ m}. Like the HL version, but operating on the designed program of a pRHL judgement’s conclusion.

\textbf{Syntax:} \texttt{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.

\textbf{Syntax:} \texttt{kill c \ \dagger \ m}. Fails unless \( n \geq 1 \) and \( m \geq 0 \). 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

$\text{kill } 2 ! 2.$

produces the goals

Type variables: <none>

Context : M.f
Bound : $[=] 1\%r$
pren = 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:** $\text{kill} \{1\} c ! m | \text{kill} \{2\} c ! m$. 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>
then running

\texttt{kill(2) 2 ! 2.}

produces the goals

Type variables: <none>

Context : M.f
Bound : [=] 1%r
pre = true
(1) M.x \leftarrow 0
(2) M.x \leftarrow M.x + 1
post = true

and

Type variables: <none>

\texttt{kill(2) c} | \texttt{kill(1)c} | \texttt{kill(2)c}. Like the general cases, but with \( m = 1 \).

\textbf{Syntax: \texttt{symmetry}}. In PRHL, swaps the two programs, transforming the pre and postconditions by swapping the memories they refer to.
CHAPTER 3. TACTICS

symmetry

\[ \begin{align*}
  \{P^{-1}\} c_2 & \sim c_1 \{Q^{-1}\} \\
  \{P\} c_1 & \sim c_2 \{Q\}
\end{align*} \]

transitivity

**Syntax:** transitivity \( c \) \((P_1 \Rightarrow Q_1) \) \((P_2 \Rightarrow Q_2)\). In pRHL, applies the transitivity of program equivalence using the specified program and specifications. When the goal is a judgment on procedures, \( c \) should be a procedure. When the goal is a judgment on statements, \( c \) should be a statement, and the tactic then takes a side argument, used to decide the procedure context under which local variables from \( c \) are evaluated.

**Examples:**

\[
\text{transitivity} \, f \, ([P_1 \Rightarrow Q_1] \, [P_2 \Rightarrow Q_2])
\]

\[
\forall m_1 \ m_2. \ m_1 \ P \ m_2 \Rightarrow \exists m. \ m_1 \ P_1 \ m \land \ m \ P_2 \ m_2 \\
\forall m_1 \ m_2. \ m_1 \ Q_1 \ m \Rightarrow m \ Q_2 \ m_2 \Rightarrow m_1 \ Q \ m_2 \quad \{P_1\} \ f_1 \sim f \{Q_1\} \\
\{P_2\} \ f \sim f_2 \{Q_2\}
\]

\[
\text{transitivity} \, \{s\} \, ([P_1 \Rightarrow Q_1] \, [P_2 \Rightarrow Q_2])
\]

\[
\forall m_1 \ m_2. \ m_1 \ P \ m_2 \Rightarrow \exists m. \ m_1 \ P_1 \ m \land \ m \ P_2 \ m_2 \\
\forall m_1 \ m_2. \ m_1 \ Q_1 \ m \Rightarrow m \ Q_2 \ m_2 \Rightarrow m_1 \ Q \ m_2 \quad \{P_1\} \ s_1 \sim s \{Q_1\} \\
\{P_2\} \ s \sim s_2 \{Q_2\}
\]

**Note:** In practice, the existential quantification over memory \( m \) in the first generated subgoal is replaced with an existential quantification over the program variables appearing in \( P, P_1 \), or \( P_2 \).

conseq

**Syntax:** conseq \( (_:\, P \Rightarrow Q) \) | conseq \( (_:\, P \Rightarrow Q: \delta) \). Prove a program logic judgment by weakening a stronger result (rule of consequence). Any one of the specification places can be filled with a wildcard \( (_) \) to keep its current value and trivially discharge the corresponding subgoal.

**Examples:**

\[
\text{conseq} \, (_:\, P \Rightarrow Q) \quad \text{[pRHL]}
\]

\[
\begin{align*}
  P' \Rightarrow P \\
  Q \Rightarrow Q' \\
  \{P\} \ c \sim c' \{Q\}
\end{align*}
\]

\[
\text{conseq} \, (_:\, P \Rightarrow Q: \delta) \quad \text{[pHL]}
\]

\[
\begin{align*}
  P' \Rightarrow \delta \circ \delta' \\
  P' \Rightarrow P \\
  Q \circ Q' \\
  \{P\} \ c \{Q\} \circ \delta
\end{align*}
\]

\[
\text{conseq} \, (_:\, P \Rightarrow Q) \quad \text{[HL]}
\]

\[
\begin{align*}
  P' \Rightarrow P \\
  Q \Rightarrow Q' \\
  \{P\} \ c \{Q\}
\end{align*}
\]

**Note:** The pHL variant can also be used to strengthen the relation \( \circ \) into an equality by forcing the equality into the specification. For example, the following is a valid application of conseq.
CHAPTER 3. TACTICS

\[
\text{Syntax: } \text{conseq } H. \text{ Only applies to judgments on procedures. Same as conseq with a specification, but the specification to use is inferred from the lemma } H \text{ provided. Raises an error if the lemma does not refer to the expected procedure(s). All variants of conseq may take lemmas in place of explicit specifications with the same effect, in which case they must be applied to judgments on procedures.}
\]

\[
\text{Syntax: conseq*. Same as conseq, but the subgoal corresponding to the postcondition is refined by a “may modify” analysis. All variants of conseq can be refined using the *, with the same effect.}
\]

\[
\text{Syntax: conseq } \langle \text{prhl} \rangle \langle \text{hl} \rangle \langle \text{hl} \rangle. \text{ Combine relational and non-relational specifications to prove a relational specification. Either one of the Hoare logic specifications can be replaced with a wildcard.}
\]

\[
\text{Examples:}
\]

\[
\text{Syntax: case } e. \text{ Split the current pRHL, pHL or HL goal by doing a case analysis in the precondition.}
\]

\[
\text{Examples:}
\]
CHAPTER 3. TACTICS

case $C$ \hspace{1cm} [pRHL]
\[
\frac{\{P \wedge C\} \; c_1 \sim c_2 \; \{Q\} \quad \{P \wedge \neg C\} \; c_1 \sim c_2 \; \{Q\}}{\{P\} \; c_1 \sim c_2 \; \{Q\}}
\]

\begin{align*}
\text{case } C & \quad \text{[pHL]} & \text{case } C & \quad \text{[HL]} \\
\{P \wedge C\} \; c \{Q\} \circ \delta & \quad \{P \wedge \neg C\} \; c \{Q\} \circ \delta & \quad \{P \wedge C\} \; c \{Q\} \\
\{P\} \; c \{Q\} \circ \delta & \quad \{P \wedge \neg C\} \; c \{Q\} \\
\end{align*}

\textbf{phoare split}

\textbf{Syntax}: \texttt{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.

\textbf{Examples:}

\begin{align*}
\text{phoare split } & \delta_A \; \delta_B \; \delta_{AB} \quad \text{[pHL]} \\
\delta_A + & \delta_B - \delta_{AB} \circ \delta \\
\{P\} \; c \{A\} \circ & \delta_A \quad \{P\} \; c \{B\} \circ & \delta_B \quad \{P\} \; c \{A \wedge B\} \circ^{-1} \delta_{AB} \\
\{P\} \; c \{A \lor B\} \circ & \delta
\end{align*}

\begin{align*}
\text{phoare split } & \delta_A \; \delta_B \; \delta_{AB} \quad \text{[pHL]} \\
\delta_A + & \delta_B - \delta_{AB} \circ \delta \\
\{P\} \; c \{A\} \circ & \delta_A \quad \{P\} \; c \{B\} \circ & \delta_B \quad \{P\} \; c \{A \lor B\} \circ^{-1} \delta_{AB} \\
\{P\} \; c \{A \land B\} \circ & \delta
\end{align*}

\textbf{Syntax:} \texttt{phoare split }$! \; \delta_T \; \delta_i$. Splits a pHL judgment into two judgments whose postcondition are true and the negation of the original postcondition, respectively.

\textbf{Examples:}

\begin{align*}
\text{phoare split } & ! \; \delta_T \; \delta_i \quad \text{[pHL]} \\
\delta_T - & \delta_i \circ \delta \\
\{P\} \; c \{\text{true}\} \circ & \delta_T \quad \{P\} \; c \{\neg Q\} \circ^{-1} \delta_i \\
\{P\} \; c \{Q\} \circ & \delta
\end{align*}

\textbf{Syntax:} \texttt{phoare split }$\delta_A \; \delta_{A}: \lambda$. Splits a pHL judgment following an event $A$.

\textbf{Examples:}

\begin{align*}
\text{phoare split } & \delta_A \; \delta_{A}: \lambda \quad \text{[pHL]} \\
\delta_A + & \delta_{A} \circ \delta \\
\{P\} \; c \{Q \wedge A\} \circ & \delta_A \quad \{P\} \; c \{Q \wedge \neg A\} \circ & \delta_{A}
\end{align*}

\textbf{byequiv}

\textbf{Syntax}: \texttt{byequiv }[\text{option}]? <spec>. Derives a probability relation from a pRHL judgement on the procedures involved. <spec> can include wildcards when the tactic should infer the pre or postcondition. In addition, <spec> can be extended with a failure event to infer precise applications of the Fundamental Lemma.

\textbf{Options}: By default, (eq option) specification inference attempts to infer a conjunction of equalities sufficient to imply the desired relation. Passing the ~eq option overrides this behaviour, instead using the trivial relation on events.
Examples:

\begin{align*}
\text{byequiv } (_:: P \Rightarrow Q) \\
\{P\} f _1 \sim f _2 \{Q\} \\
m_1[\text{arg} \mapsto \vec{a}_1] P m_2[\text{arg} \mapsto \vec{a}_2] \quad Q \Rightarrow E_1[1] \Leftrightarrow E_2[2] \\
\Pr[m_1, f_1(\vec{a}_1) : E_1] = \Pr[m_2, f_2(\vec{a}_2) : E_2]
\end{align*}

\begin{align*}
\text{byequiv } (_:: P \Rightarrow Q) \\
\{P\} f _1 \sim f _2 \{Q\} \\
m_1[\text{arg} \mapsto \vec{a}_1] P m_2[\text{arg} \mapsto \vec{a}_2] \quad Q \Rightarrow E_1[1] \Rightarrow E_2[2] \\
\Pr[m_1, f_1(\vec{a}_1) : E_1] \leq \Pr[m_2, f_2(\vec{a}_2) : E_2]
\end{align*}

\begin{align*}
\text{byequiv } (_:: P \Rightarrow Q) \\
\{P\} f _1 \sim f _2 \{Q\} \\
m_1[\text{arg} \mapsto \vec{a}_1] P m_2[\text{arg} \mapsto \vec{a}_2] \quad Q \Rightarrow E_2[2] \Rightarrow E_1[1] \\
\Pr[m_1, f_1(\vec{a}_1) : E_1] \geq \Pr[m_2, f_2(\vec{a}_2) : E_2]
\end{align*}

\begin{align*}
\text{byequiv } (_:: P \Rightarrow Q) : B_1 \\
m_1[\text{arg} \mapsto \vec{a}_1] P m_2[\text{arg} \mapsto \vec{a}_2] \quad Q \Rightarrow \neg B_2[2] \Rightarrow E_1[1] \Rightarrow E_2[2] \\
\Pr[m_1, f_1(\vec{a}_1) : E_1] \leq \Pr[m_2, f_2(\vec{a}_2) : E_2] + \Pr[m_2, f_2(\vec{a}_2) : B_2]
\end{align*}

\begin{align*}
\text{byequiv } (_:: P \Rightarrow Q) \\
\{P\} f _1 \sim f _2 \{E_1[1] \Rightarrow E_2[2]\} \\
m_1[\text{arg} \mapsto \vec{a}_1] P m_2[\text{arg} \mapsto \vec{a}_2] \quad Q \Rightarrow (B_1[1] \Leftrightarrow B_2[2]) \land (\neg B_2[2] \Rightarrow E_1[1] \Rightarrow E_2[2]) \\
|\Pr[m_1, f_1(\vec{a}_1) : E_1] - \Pr[m_2, f_2(\vec{a}_2) : E_2]| \leq \Pr[m_2, f_2(\vec{a}_2) : B_2]
\end{align*}

\begin{align*}
\text{byequiv } [\text{eq}] (_:: P \Rightarrow _) \\
\{P\} f _1 \sim f _2 \{\text{E}_1[1] \Leftrightarrow \text{E}_2[2]\} \\
m_1[\text{arg} \mapsto \vec{a}_1] P m_2[\text{arg} \mapsto \vec{a}_2] \\
\Pr[m_1, f_1(\vec{a}_1) : E_1] = \Pr[m_2, f_2(\vec{a}_2) : E_2]
\end{align*}

\textbf{Syntax: \texttt{byequiv \langle lemma\rangle}.} Same as \texttt{byequiv <spec>}, but the specification to use is inferred from the lemma provided. Raises an error if the lemma does not refer to the expected procedures. Inference options have no effect in this setting.

\textcopyright \texttt{byphoare}

\textbf{Syntax: \texttt{byphoare [option]? <spec>}.} Derives a probability relation from a PHL judgement on the procedure involved. \texttt{<spec> can include wildcards when the tactic should infer the pre or postcondition.}

\textbf{Options:} By default, \texttt{(eq option)} specification inference attempts to infer a conjunction of equalities sufficient to imply the desired relation. Passing the \texttt{~eq option} overrides this behaviour, instead using the trivial relation on events.

\textbf{Examples:}

\begin{align*}
\text{byphoare } (_:: P \Rightarrow Q) \\
\{P\} f \{Q\} = \delta \\
\text{P } m[\text{arg} \mapsto \vec{a}] \quad \forall m', Q m' \Leftrightarrow E m' \\
\Pr[m, f(\vec{a}) : E] = \delta
\end{align*}
Syntax: `byphoare <lemma>`. Same as `byphoare <spec>`, but the specification to use is inferred from the lemma provided. Raises an error if the lemma does not refer to the expected procedure. Inference options have no effect in this setting.

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. 126).

Examples:

\[
\begin{align*}
\text{hoare} & \quad \text{hoare} \\
\{ \text{true} \} & f \{ \neg \neg \} \\
\Pr[m, f(a) : \neg \neg] & = 0 \\
\{ \text{P} \} & f \{ \neg \neg \} \\
\{ \text{P} \} & f \{ \neg \neg \} \leq 0
\end{align*}
\]

Syntax: `bypr` Derives a program judgment from a probability relation or an exact probability. Only applies to judgments on procedures.

Examples:

\[
\begin{align*}
\text{bypr} & \quad (r_1) \ (r_2) \\
\forall m_1, m_2, a, r_1 = a & \Rightarrow r_2 = a \Rightarrow m_1 \ Q \ m_2 \\
\forall \bar{a}_1, \bar{a}_2, m_1, m_2, a, m_1 & [\text{arg} \mapsto \bar{a}_1] \ P \ m_2 [\text{arg} \mapsto \bar{a}_2] \Rightarrow \\
\Pr[m_1, f_1(\bar{a}_1) : a = r_1] & = \Pr[m_2, f_2(\bar{a}_2) : a = r_2] \\
\{ \text{P} \} & f_1 \sim f_2 \ {\text{Q}}
\end{align*}
\]

Syntax: `exists*` Introduce an existential quantification over the value of a program variable in the initial memory. This is particularly useful when dealing with a procedure call using a lemma that refers to initial values of arguments or state (using `call` (p. 93)). Several program variables can be treated simultaneously by providing them in a comma-separated list.

Examples:
CHAPTER 3. TACTICS

exists* M.x(1) \[ pRHL \]
\[
\exists x, x = M.x(1) \land P \vdash c_1 \sim c_2 \{Q\}
\]
\[
\{P\} c_1 \sim c_2 \{Q\}
\]

exists* M.x(1), M.x(2) \[ pRHL \]
\[
\exists x_1, x_2, x_1 = M.x(1) \land M.x(2) \land P \vdash c_1 \sim c_2 \{Q\}
\]
\[
\{P\} c_1 \sim c_2 \{Q\}
\]

exists* M.x \[ pHL \]
\[
\exists x, x = M.x \vdash c_1 \sim c_2 \{Q\}
\]
\[
\{P\} c_1 \sim c_2 \{Q\} \circ \delta
\]

exists* M.x \[ HL \]
\[
\exists x, x = M.x \vdash c_1 \sim c_2 \{Q\}
\]
\[
\{P\} c_1 \sim c_2 \{Q\}
\]

\[ \circ \] elim*
Destruct existential quantifications at the head of a precondition. Such existential quantifications may be introduced by sp (p. 81) or exists* (p. 130).

Examples:

\[ \circ \] elim* \[ pRHL \] elim* \[ pHL \]
\[
\forall x, \{x = M.x \land P\} c_1 \sim c_2 \{Q\}
\]
\[
\exists x, x = M.x \land P \vdash c_1 \sim c_2 \{Q\}
\]
\[
\forall x, \{x = M.x \land P\} c_1 \sim c_2 \{Q\} \circ \delta
\]
\[
\exists x, x = M.x \land P \vdash c_1 \sim c_2 \{Q\} \circ \delta
\]

\[ \circ \] elim* \[ HL \]
\[
\forall x, \{x = M.x \land P\} c_1 \sim c_2 \{Q\}
\]
\[
\exists x, x = M.x \land P \vdash c_1 \sim c_2 \{Q\}
\]

\[ \circ \] exfalso
Syntax: exfalso. Combines conseq (p. 126), byequiv (p. 128), byphoare (p. 129), hoare (p. 130) and bypr (p. 130) to strengthen the precondition into false and discharge the resulting trivial goal.

Examples:

\[ \circ \] exfalso \[ pRHL \] exfalso \[ pHL \] exfalso \[ HL \]
\[
P \Rightarrow false
\]
\[
\{P\} c_1 \sim c_2 \{Q\}
\]
\[
P \Rightarrow false
\]
\[
\{P\} c_1 \sim c_2 \{Q\} \circ \delta
\]
\[
P \Rightarrow false
\]
\[
\{P\} c_1 \sim c_2 \{Q\} \circ \delta
\]

FiXme Note: Move exfalso to automatic tactics?

3.3.4 Automated Tactics

\[ \circ \] auto
FiXme Note: Missing description of auto.

\[ \circ \] sim
Syntax: `\texttt{sim} \; \texttt{<pos>}? \; \texttt{<hintgeqs>}* \; \texttt{<hintinv>}? \; \texttt{<eqs>}?.$

where \texttt{<pos>} = \texttt{<uint>} \; \texttt{<uint>}

\texttt{<hintgeqs>} = (\langle \texttt{procname}\rangle? \sim \langle \texttt{procname}\rangle? : \langle \texttt{formula}\rangle)

| \langle \_\_? : \langle \texttt{formula}\rangle |

\texttt{<eqs>} = : \langle \texttt{formula}\rangle

\textbf{Fixme} Note: Missing description of sim.

### 3.3.5 Advanced Tactics

- **\texttt{eager}**

\textbf{Fixme} Note: Missing description of eager. \textbf{Fixme} Note: Missing descriptions for all eager `<tactic>` variants.

- **\texttt{fel}**

\textbf{Fixme} Note: Missing description of fel.
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


Index of Tactics

admit, 58
algebra, 68
alias, 120
apply, 59
assumption, 47
auto, 131
byequiv, 128
byphoare, 129
bypr, 130
call, 93
case, 63
case-pl, 127
cfold, 122
change, 58
clear, 47
closing goals, 71
congr, 52
conseq, 126
cut, 59
done, 55
eager, 132
elim, 67
elim*, 131
exact, 61
exfalso, 131
exists, 49
exists*, 130
failure recovery, 69
fel, 132
fission, 117
fusion, 119
generalization, 45
goal selection, 70
have, 59
hoare, 130
idtac, 47
if, 85
inline, 106
introduction, 40
kill, 123
left, 48
move, 47
phoare split, 128
pose, 58
proc, 72
progress, 56
rcondf, 110
rcondt, 108
reflexivity, 48
rewrite, 61
right, 48
rnd, 83
seq, 79
sequence, 68
sequence with branching, 69
sim, 131
simplify, 55
skip, 79
smt, 57
sp, 81
split, 50
splitwhile, 115
subst, 53
swap, 103
symmetry, 125
tactic repetition, 69
transitivity, 126
trivial, 55
unroll, 112
while, 89
wp, 82
List of Corrections

Note: Need explanation of how a proof term may be used in forward reasoning. . . . . 39
Note: Can it be used in forward reasoning? . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
Note: In forward reasoning they aren’t equivalent—why? . . . . . . . . . . . . . . . . 39
Note: Be a bit more detailed about what this tactic does? . . . . . . . . . . . . . . . . . 39
Note: Is this the right place to define “convertible”? . . . . . . . . . . . . . . . . . . . . 56
Note: Describe progress options. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
Note: Describe failure states of prover selection. . . . . . . . . . . . . . . . . . . . . . . . . . . 57
Note: Describe dbhint options. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
Note: Make this a pragma? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
Note: Missing description of algebra . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
Note: Move exfalso to automatic tactics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131
Note: Missing description of auto . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131
Note: Missing description of sim . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132
Note: Missing description of eager . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132
Note: Missing descriptions for all eager <tactic> variants . . . . . . . . . . . . . . . . . . . . . . 132
Note: Missing description of fel . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132