EASYCRYPT Reference Manual

Version 1.x — Compiled on February 18, 2018
# Contents

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

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

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

5 EASYCRYPT Library ........................................ 201
Chapter 1

Getting Started

1.1 Introduction

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

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

1.2 Installing EasyCrypt

EASYCRYPT may be found on GitHub.

https://github.com/EasyCrypt/easycrypt

Detailed building instructions for EASYCRYPT and its dependencies and supporting tools can be found in the project’s README file.1

1.3 Running EasyCrypt

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

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

---

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

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

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

A sample EasyCrypt script is shown in Listing 1.1.

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

require import Bool DBool.

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

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

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

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

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

lemma G1_G2_f : equiv[ G1.f ~ G2.f : true => ={res}].

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

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

lemma G1_G2_true &m :
  Pr[G1.f() @ &m : res] = Pr[G2.f() @ &m : res].

proof.
  byequiv.
  apply G1_G2_f. trivial. trivial.
As can be inferred from the example, comments begin and end with (* and *), respectively; they may be nested. Each sentence of an EASYCRYPT script is terminated with a dot (period, full stop). Much can be learned by experimenting with this script, and in particular by executing it step-by-step in Emacs.

### 1.4 More Information

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

https://www.easycrypt.info

The EASYCRYPT Club mailing list features discussion about EASYCRYPT usage:

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

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

### 1.5 Bug Reporting

EASYCRYPT bugs should be reported using the Tracker:

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

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

### 1.6 About this Documentation

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

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

²https://github.com/EasyCrypt/easycrypt-doc
Chapter 2

Specifications

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

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

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

In its expression language:

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

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

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

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

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

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

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

## 2.1 Lexical Categories

EASYCRYPT’s language has a number of lexical categories:

- **Keywords.** EASYCRYPT has the following keywords: abbrev, abort, abstract, admit, admitted, algebra, alias, apply, as, assert, assumption, async, auto, axiom, axiomatized, beta, by, byequiv, byhoare, bypr, call, case, cfold, change, class, clear, clone, congr, conseq, const, cut, debug, declare, delta, do, done, dump, eager, elif, elim, else, end, equiv, eta, exact, exfalso, exists, expect, export, fel, field, fieldeq, first, fission, forall, fun, fusion, glob, goal, have, hint, hoare, idtac, if, import, in, include, inductive, inline, instance, iota, islossless, kill, last, left, lemma, let, local, logic, modpath, module, move, nomm, notation, of, op, phoare, pose, Pr, pr_bounded, pragma, pred, print, proc, progress, proof, prover, qed, rcondf, rcondt, realize, reflexivity, remove, rename, replace, require, res, return, rewrite, right, ring, ringeq, rnd, rwnormal, search, section, Self, seq, sim, simplify, skip, smt, solve, sp, split, splitwhile, strict, subset, suff, swap, symmetry, then, theory, time, timeout, Top, transitivity, trivial, try, type, undo, unroll, var, while, why3, with, wlog, wp and zeta.

- **Identifiers.** An identifier is a sequence of letters, digits, underscores (_) and apostrophes (’’) that begins with a letter or underscore, and isn’t equal to an underscore or a keyword other than abort, admitted, async, dump, expect, field, fieldeq, first, last, left, right, ring, ringeq, solve, strict or wlog.

- **Operator names.** An operator name is an identifier, a binary operator name, a unary operator name, or a mixfix operator name.

- **Binary operator names.** A binary operator name is:

  - a nonempty sequence of equal signs (=), less than signs (<), greater than signs (>), forward slashes (/), backward slashes (\), plus signs (+), minus signs (-), times signs (*), vertical bars (|), colons (:), ampersands (#), up arrows (^) and percent signs (%); or

  - a backtick mark (’), followed by a nonempty sequence of one of these characters, followed by a backtick mark; or

  - a backward slash followed by a nonempty sequence of letters, digits, underscores and apostrophes.

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

- it is neither << nor >>; and
- it doesn’t contain a colon, unless it is a sequence of colons of length at least two; and
- it doesn’t contain =>, except if it is =>; and
- it doesn’t contain |, except if it is ||; and
- it doesn’t contain /, except if it is /, /\, or a sequence of slashes of length at least 3.
The precedence hierarchy for infix operators is (from lowest to highest):

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- **Memory identifiers.** A *memory identifier* consists an ampersand followed by either a nonempty sequence of digits or an identifier whose initial character isn’t an uppercase letter.
2.2 Script Structure, Name spaces, Printing and Searching

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

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

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

To print an entity, one may say:

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

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

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

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

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

2.3 Expressions Language

2.3.1 Type Expressions

EasyCrypt’s type expressions are built from type variables, type constructors (or named types) function types and tuple (product) types. Type constructors include built-in types and user-defined types, such as record types and datatypes (or variant types). The syntax of type expressions
\[\begin{align*}
\tau, \sigma & ::= \text{tyvar} & \text{type variable} \\
\tau & ::= \tau, \sigma & \text{anonymous type variable} \\
(\tau) & ::= \text{parenthesized type} \\
\tau \rightarrow \sigma & ::= \text{function type} \\
(\tau_1 \ast \cdots \ast \tau_n) & ::= \text{tuple type} \\
\text{tyname} & ::= \text{named type} \\
\tau \text{ tyname} & ::= \text{applied type constructor} \\
(\tau_1, \ldots, \tau_n) \text{ tyname} & ::= \text{ibid.}
\end{align*}\]

Figure 2.1: EasyCrypt’s type expressions

<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>(\rightarrow)</td>
<td>function type</td>
</tr>
</tbody>
</table>

constructions with higher precedences come first

Figure 2.2: Type operators precedence and associativity

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

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

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

In addition, to every type \(t\) is associated the type \(t\text{ distr}\) 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 \(\text{int} \rightarrow \text{bool} \rightarrow \text{real}\) and \(\text{int} \rightarrow (\text{bool} \rightarrow \text{real})\) 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 ‘\(a \ast a\)’ 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 \text{map} of arity 2.

By abuse of notations, named types (as \text{bool} or \text{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:

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

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

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

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

Datatypes and datatype operators may be declared like this:

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

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

```haskell
  type tree = [
    | Leaf of int
```
\begin{verbatim}
| Cons of tree & int & tree |.
\end{verbatim}

\textit{Cons} \(tr_1 \times tr_2\) 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{verbatim}
type t = int * bool.
type ('a, b) arr = 'a -> 'b.
\end{verbatim}

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

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

\begin{verbatim}
type t.
type ('a, b) u.
type t, ('a, b) u.
\end{verbatim}

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

### 2.3.3 Expressions and Operator Declarations

We'll now survey \textsc{EasyCrypt}'s typed expressions. Anonymous functions are written

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

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 \(t_1 \to t_2\).

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

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

or

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

or (letting \textsc{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.

\textsc{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 \textsc{EasyCrypt} infer them.

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

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

or

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

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

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

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

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

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

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

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

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

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

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

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

One may declare multiple abstract operators of the same type:

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

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

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

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

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

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

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

\[
\text{op } y' : \text{bool} = \text{true} -- 0.
\]
then x and x’ evaluate to 0, and y and y’ evaluate to true.

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

\[
\begin{align*}
\text{op } & x : \text{int}. \\
\text{op } & f : \text{int} \rightarrow \text{int}. \\
\text{op } & [!] : \text{int} \rightarrow \text{int}. \\
\text{op } & y : \text{int} = f \ x. \\
\text{op } & y’ : \text{int} = [!] (f \ x).
\end{align*}
\]

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).

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

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

then x will evaluate to 3.

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

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

then y will evaluate to the same value as x.

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

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

then y will evaluate to the same value as x.

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

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

then y will evaluate to the same value as x.

In addition, if \(e_1, \ldots, e_n\) are expressions then

\[
[e_1; \ldots; e_n] \quad \text{is translated to} \quad e_1 :: \ldots :: e_n :: []
\]

whenever the latter expression is well-typed. The initial argument of "[ ]" and "[<= ]" have higher precedence than even function application. E.g., one can’t omit the parentheses in
Some operators are built-in to EASYCRYPT, automatically understood by its ambient logic:

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

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

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

If \( e_1 \) is an expression of type bool and \( e_2, e_3 \) are expressions of some type \( t \), then the conditional expression

```plaintext
e_1 ? e_2 : e_3
```

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

```plaintext
if e_1 then e_2 else e_3
```

E.g., if we write

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

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

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

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

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

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

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

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

we’ll end up with z evaluating to false. One may combine requiring and importing in one step:
require import Bool Int Real Distr.
We’ll cover theories and their usage in detail in Chapter 4.

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

Values of product (tuple) and record types are constructed and destructed as follows:
op x : int * int * bool = (3, 4, true).
op b : bool = x.
type t = { u : int; v : bool; }.
op y : t = { v = false; u = 10; }.
op a : bool = y.

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

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

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

then z evaluates to the same result as x.

We can declare operators using pattern matching on the constructors of datatypes. E.g., continuing the previous example, we can declare and use an operator fst by:
op fst ['a, 'b] (def : 'a) (ei : ('a, 'b) either) : 'a =
with ei = Fst a => a
with ei = Snd b => def.
op l1 : (int, bool) either = Fst 10.
op l2 : (int, bool) either = Snd true.
op m1 : int = fst (-1) l1.
op m2 : int = fst (-1) l2.

Here, m1 will evaluate to 10, whereas m2 will evaluate to -1. Such operator declarations may be
recursive, as long as EASYCRYPT can determine that the recursion is well-founded. E.g., here is
one way of declaring an operator length that computes the length of a list:
type 'a list = [Nil | Cons of 'a & 'a list].
op len ['a] (acc : int, xs : 'a list) : int =
with xs = Nil => acc
with xs = Cons y ys => len (acc + 1) ys.
op length ['a] (xs : 'a list) = len 0 xs.
op xs = Cons 0 (Cons 1 (Cons 2 Nil)).
op n : int = length xs.

Then n will evaluate to 3.
2.4 Module System

2.4.1 Modules

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

```plaintext
require import Int Bool DInterval.

module M = {
  var x : int

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

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

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

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

Listing 2.1: Simple Module

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

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

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

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

- It declares a local variable, $n$, of type int—local in the sense that other procedures can’t access or affect it.
- It uses a procedure call to call the procedure init with a bound of 100, causing $x$ to be initialized to an integer between -100 and 100.
- It calls incr twice, with 10 and then -50.
- It uses a procedure call assignment to call the procedure get with no arguments, and assign get’s return value to $n$.
- It evaluates the boolean expression $n < 0$, and returns the value of this expression as its boolean result.
EASYCRYPT tries to infer the return types of procedures and the types of parameters and local variables. E.g., our example module could be written

```
module M = {
  var x : int
  proc init(bnd) = {
    x <$> [-bnd .. bnd];
  }
  proc incr(n) = {
    x <- x + n;
  }
  proc get() = {
    return x;
  }
  proc main() = {
    var n;
    init(100);
    incr(10); incr(-50);
    n <$> get();
    return n < 0;
  }
};
```

Listing 2.2: Simple Module with Type Inference

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

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

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

- The rhs of a random assignment must be a single (sub-)distribution. When choosing from a proper sub-distribution, the random assignment may fail, causing the procedure call that invoked it to fail to terminate.
- The rhs of an ordinary assignment may be an arbitrary expression (which doesn’t include use of procedures).
- the rhs of a procedure call assignment must be a single procedure call.

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

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

in the case where ... produces a triple.

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

```
require import Bool Int DInterval.

module N = {
```
\[
\begin{align*}
\text{proc} \ \text{loop()} : \text{int} = \{ \\
\text{var} \ y, z : \text{int}; \\
y \leftarrow 0; \\
\text{while} \ (y < 10) \{ \\
z <\{1 .. 10\}; \\
\text{if} \ (z \leq 5) \{ \\
y \leftarrow y - z; \\
\} \\
\text{else} \{ \\
y \leftarrow y + (z - 5); \\
\} \\
\} \\
\text{return} \ y;
\}
\end{align*}
\]

Listing 2.3: Conditionals and While Loops

\(N\) has a single procedure, \text{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:

\[
\begin{align*}
\text{if} \ (z \leq 5) \ y \leftarrow y - z; \\
\text{else} \ y \leftarrow y + (z - 5);
\end{align*}
\]

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

\[
\begin{align*}
\text{if} \ (z \leq 5) \ y \leftarrow y - z;
\end{align*}
\]

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

\[
\begin{align*}
\text{require import \ Bool \ Int.} \\
\text{module M = \{} \\
\text{var x : int} \\
\text{proc f() : unit = \{} \\
x \leftarrow x + 1; \\
\} \\
\}. \\
\text{module N = \{} \\
\text{var x : int} \\
\text{proc g(n m : int, b : bool) : bool = \{} \\
\text{if} \ (b) \ M.f(); \\
M.x \leftarrow M.x + x + n - m; \\
\text{return} \ 0 < M.x; \\
\}\}
\end{align*}
\]
Procedure $g$ of $N$ both accesses the global variable $x$ of module $M$ ($M.x$), and calls $M$’s procedure, $f$ ($M.f$). The parameter list of $g$ could equivalently be written:

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

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

```
proc $f() : unit =$
M.x $\leftarrow M.x + 1$
```

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

```
module $L = N$
```

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

```
module $X =$
proc $f() : bool =$
var $b : bool$
return $b$

\}
```

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

```
module $Y =$
proc $f() : int =$
var $x : int$
return $x - x$

\}
```

### 2.4.2 Module Types

EASYCRYPT’s module types specify the types of a set of procedures. E.g., consider the module type $\text{OR}$:

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

\}
```

$\text{OR}$ describes minimum expectations for a “guessing oracle”—that it provide at least procedures with the specified types. The order of the procedures in a module type is irrelevant. In a procedure’s type, one may combine multiple parameters of the same type, as in:

```
proc $\text{init}(\text{secret \ tries} : \text{int}) : \text{unit}$
The names of procedure parameters used in module types are purely for documentation purposes; one may elide them instead using underscores, writing, e.g.,

\[
\text{proc init}(_ : \text{int}, _ : \text{int}) : \text{unit}
\]

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.

\[
\begin{align*}
\text{module } \text{Or} & = \{ \\
\text{var } \text{sec} : \text{int} \\
\text{var } \text{tris} : \text{int} \\
\text{var } \text{guessed} : \text{bool} \\
\text{proc } \text{init}(\text{secret, tries} : \text{int}) : \text{unit} = \{ \\
\text{sec} & \leftarrow \text{secret}; \\
\text{tris} & \leftarrow \text{tries}; \\
\text{guessed} & \leftarrow \text{false}; \\
\} \\
\text{proc } \text{guess}(\text{guess} : \text{int}) : \text{unit} = \{ \\
\text{if } (0 < \text{tris}) \{ \\
\text{guessed} & \leftarrow \text{guessed} \lor (\text{guess} = \text{sec}); \\
\text{tris} & \leftarrow \text{tris} - 1; \\
\} \\
\} \\
\text{proc } \text{guessed}() : \text{bool} = \{ \\
\text{return } \text{guessed}; \\
\} \\
\}
\end{align*}
\]

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. \texttt{Or} satisfies the specification of the module type \texttt{OR}, and we can ask \textsc{EasyCrypt} to check this by supplying that module type when declaring \texttt{Or}, as in Listing 2.6.

\[
\begin{align*}
\text{module } \text{Or} : \text{OR} & = \{ \\
\text{var } \text{sec} : \text{int} \\
\text{var } \text{tris} : \text{int} \\
\text{var } \text{guessed} : \text{bool} \\
\text{proc } \text{init}(\text{secret, tries} : \text{int}) : \text{unit} = \{ \\
\text{sec} & \leftarrow \text{secret}; \\
\text{tris} & \leftarrow \text{tries}; \\
\text{guessed} & \leftarrow \text{false}; \\
\} \\
\text{proc } \text{guess}(\text{guess} : \text{int}) : \text{unit} = \{ \\
\text{if } (0 < \text{tris}) \{ \\
\text{guessed} & \leftarrow \text{guessed} \lor (\text{guess} = \text{sec}); \\
\text{tris} & \leftarrow \text{tris} - 1; \\
\} \\
\}
\end{align*}
\]
CHAPTER 2. SPECIFICATIONS

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:

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

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

```plaintext
module X’ : A, B = X.
```

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

```plaintext
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

```plaintext
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

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

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

```plaintext
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:

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

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

The full Guessing Game example is contained in Listing 2.7.

```plaintext
require import Bool Int IntDiv DInterval.

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

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

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

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

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

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

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

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

  proc doGuessing() : unit = {
    var x : int;
    while (0 < tries) {
      ```
x <$ [range.'1 .. range.'2];
0.guess(x);
tries <- tries - 1;
}
}
}.

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

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

module SimpGame : GAME = Game(SimpAdv).

Listing 2.7: Full Guessing Game Example

SimpAdv is a simple implementation of an adversary. The inclusion of the constraint SimpAdv : ADV in SimpAdv’s declaration makes EASYCRYPT check that SimpAdv implements the module type ADV: its implementation of chooseRange doesn’t use 0 at all; its implementation of doGuessing doesn’t use any of 0’s procedures other than guess; and that chooseRange initializes SimpAdv’s global variables. Its chooseRange procedure chooses the range of 1 to 100, and initializes global variables recording this range and the number of guesses it will make (see the code of Game to see why 10 is a sensible choice). The doGuessing procedure makes 10 random guesses. It would be legal for the parameter 0 in SimpAdv’s definition to be constrained to match a module type T providing a proper subset of 0’s procedures, but that would further limit what procedures of 0 SimpAdv’s procedures could call. On the other hand, it would be illegal for T to provide procedures not in 0.

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

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

On the other hand, to call one of SimpAdv’s procedures, one needs to specify which oracle parameter it will use, as in:
module X = {
  proc f() : unit = {
    SimpAdv(Or).doGuessing();
  }
}. 

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

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

• Otherwise, it uses Or.init to initialize the guessing oracle with a secret that’s randomly chosen from the range, plus a number of allowed guesses that’s one tenth of the range’s size.

• It then calls A.doGuessing, allowing the adversary to attempt the guess the secret.

• Finally, it calls Or.guessed to learn whether the adversary has guessed the secret, returning this boolean value as its result.

Finally, the declaration

module SimpGame = Game(SimpAdv).

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

2.4.3 Global Variables

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

• the set of global variables that are declared in M; and

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

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

print glob M.

For example, suppose we make these declarations:

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

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, \( \text{glob} M \), where a value of type \( \text{glob} M \) 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
- \( \text{HL}, \text{pHL} \) and \( \text{pRHL} \) judgments,

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

\[
\text{op} \ [!] : \text{bool} \to \text{bool}.
\]

the two semantically equivalent disjunctions

\[
\text{op} \ (||) : \text{bool} \to \text{bool} \to \text{bool}.
\]
\[
\text{op} \ (\lor) : \text{bool} \to \text{bool} \to \text{bool}.
\]

the two semantically equivalent conjunctions

\[
\text{op} \ (\&\&) : \text{bool} \to \text{bool} \to \text{bool}.
\]
\[
\text{op} \ (\land) : \text{bool} \to \text{bool} \to \text{bool}.
\]

implication

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

and if-and-only-if

\[
\text{op} \ (\Leftrightarrow) : \text{bool} \to \text{bool} \to \text{bool}.
\]

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
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.,

\[
\begin{align*}
\forall x : t, \ y : t, \ z : bool, \ \ldots \\
\exists x : t, \ y : t, \ z : bool, \ \ldots \\
\end{align*}
\]

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:

\[
\begin{align*}
pred P_0. \\
pred P_1 : int. \\
pred P_2 : int \& (int \times bool). \\
pred P_3 : int \& (int \times bool) \& (real \rightarrow int). \\
\end{align*}
\]

\( P_0, P_1, P_2 \) and \( P_3 \) are extended expressions of types \( bool, int \rightarrow bool, int \rightarrow int \times bool \rightarrow bool \) and \( int \rightarrow int \times bool \rightarrow (real \rightarrow int) \rightarrow bool \), respectively. The parentheses are mandatory in \( int \times bool \) and \( real \rightarrow int \). Thus, if \( e_1, e_2 \) and \( e_3 \) are extended expressions of types \( int, int \times bool \) and \( real \rightarrow int \), respectively, then \( P_0, P_1 e_1, P_2 e_1 e_2 \) and \( P_3 e_1 e_2 e_3 \) are formulas.

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

\[
\begin{align*}
pred Q (x \ y : int, \ z : bool) = x = y \land z. \\
\end{align*}
\]

or

\[
\begin{align*}
pred Q (x : int) (y : int) (z : bool) = x = y \land z. \\
\end{align*}
\]

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

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

meaning that, e.g.,

\[
\begin{align*}
(\text{fun} \ (b : \text{bool} \rightarrow \text{bool}) \Rightarrow b \ \text{true}) \ (Q \ 3 \ 4) \\
\end{align*}
\]

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

\[
\begin{align*}
pred R [\ 'a, \ 'b] : (\ 'a, \ 'a \times \ 'b). \\
pred R' [\ 'a, \ 'b] (x : \ 'a, \ y : \ 'a \times \ 'b) = (y.\ 1 = x). \\
\end{align*}
\]

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:

\[
\begin{align*}
\forall \&m, \ \phi \\
\end{align*}
\]

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:

\[
\begin{align*}
\text{module} \ X \ = \ { \var x : \text{int} }. \\
\text{module} \ Y \ = \ { \var y : \text{int} }. \\
\end{align*}
\]

Then, this is a (true) formula:
forall \&m, X.x{m} < Y.y{m} => X.x{m} + 1 <= Y.y{m}

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

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

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

If $M$ is a module and \&m is a memory, then $(\text{glob } M)(m)$ 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 res or $\text{glob } M$, for a module $M$, is replaced by $u(m)$. For example,

$$(Y1.y = Y1.z => Y1.z = Y1.y)(m)$$

expands to

$$Y1.y(m) = Y1.z(m) => Y1.z(m) = Y1.y(m)$$

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

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

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 $\hat{m}$ will terminate in a final memory satisfying $\phi$. To run $M.p$:

– $\hat{m}$ 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 HL judgment is that, for all initial memories $\hat{m}$ 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 $\hat{m}$ results in termination with a memory, the restriction of that memory to $\text{res}$ and the global variables of declared modules satisfies $\psi$.

• (pHL Judgments) A pHL judgment has one of the forms

$$\text{phoare} [M.p : \phi \Rightarrow \psi] < e$$
$$\text{phoare} [M.p : \phi \Rightarrow \psi] = e$$
$$\text{phoare} [M.p : \phi \Rightarrow \psi] > e$$

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 real.
CHAPTER 2. SPECIFICATIONS

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 \( \text{pRHL} \) 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

\[
\text{running the body of } M.p \text{ in } \&m \text{ results in termination with a memory whose restriction to res and the global variables of declared modules satisfies } \psi
\]

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

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

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

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 \( \text{res} \) 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 \( \text{res} \) 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 \( \&m \) 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 \( \&2 \) is replaced by reference to \( \&n \)) such that, for all memories \( \&m \), the value assigned to \( \&m \) by \( \Pi_p \) is the sum of all the probabilities distributed to \( \&n \) by \( f \). (When \( \psi \) is an equivalence like \( \equiv \) (i.e., \( \text{res}(1) = \text{res}(2) \)), this is particularly easy to interpret.)

- \( \text{(Lossless Assertions)} \) A lossless assertion has the form

\[
islossless M.p
\]

and is simply an abbreviation for
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
  \[ \Pr[G1.f() @ &m : \text{res}] \]
  is the probability that \( G1.f() \) returns \text{true} when run in the memory \&m. (The memory is irrelevant, and the expression’s value is \( \frac{1}{2} \).)

- The HL judgement
  \[ \text{hoare}[G2.f : \text{true} ==> \text{!res}] \]
  says that, if \( G2.f() \) halts (which we know it will), then its return value will be \text{false}. (This judgement is false.)

- The pHL judgement
  \[ \text{phoare}[G2.f : \text{true} => \text{res}] = \left( \frac{1}{2} \right) \]
  says that the probability of \( G2.f() \) returning \text{true} is \( \frac{1}{2} \). (This judgement is true.)

- The pRHL judgement
  \[ \text{equiv}[G1.f - G2.f : \text{true} => \text{res}] \]
  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 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 true.)
2.5.2 Axioms and Lemmas

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

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

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

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

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

```
lemma name : \phi by [tactic].
```

or

```
lemma name : \phi by [].
```

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

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

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

or

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

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

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

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

or

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

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

```
type monoid.
op id : monoid.
op (+) : monoid -> monoid -> monoid.
axiom LeftIdentity (x : monoid) : id + x = x.
axiom RightIdentity (x : monoid) : x + id = x.
axiom Associative (x y z : monoid) : x + (y + z) = (x + y) + z.
```
Any proofs we do involving monoids will then apply to any valid instantiation of monoid, id and (+). In Chapter 4, we’ll see how to carry out such instantiations using theory cloning.

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

\[
\text{type } t. \\
\text{axiom Empty : !}(\exists (x : t), \text{true}).
\]

will allow us to prove false.

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

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

Then lemma Lossless

\[
\text{lemma Lossless } (X <: T) : \text{islossless } X.f \Rightarrow \text{islossless } G(X).g.
\]

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

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

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’

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

which has the parameters of Invar plus a memory \(\& m\), says the same thing as Invar, but using a probability expression rather than a PHL judgement.
Chapter 3

Tactics

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

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

3.1 Proof Engine

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

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

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

```
Type variables: 'a, 'b

x : 'a
x' : 'a
y : 'b
y' : 'b
eq_xx': x = x'
eq_yy': y = y'

(x, y) = (x', y')
```

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

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

Type variables: <none>

X : T{G}  
n : int  
LL: islossless X.f

pre = G.x = n  
G(X).g  
[=] 1%r  
post = G.x = n

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

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

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

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

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

this is an pHL statement judgement:

Type variables: <none>

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

Context : M.f
pre = y \%3 = x

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

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

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

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

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 a 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

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

gives rise to the goal

Type variables: 'a, 'b

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

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

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

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 : M.f

pre = y \mod 3 = x

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

post = y \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 = y mod 3 = x \land y mod 3 = 1
(1) y <= y + 4
post = y 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 = y mod 3 = x \land y mod 3 <> 1
(1) y <= y + 2
post = y mod 3 = x mod 2 + 1
\end{verbatim}

(leaving 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, y{hr} mod 3 = x \land y{hr} mod 3 = 1 => (y{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 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{3.2 Matching}

Statement patterns are an extension of statements. \textbf{Fixme Note: Add explanation of new replace tactic, which uses statement patterns.}
CHAPTER 3. TACTICS

### Blocks

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

### Statements

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

### 3.3 Ambient logic

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

#### 3.3.1 Proof Terms

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

\[
\forall (x : \text{bool}), \; x = y \Rightarrow \forall (z : \text{bool}), \; y = z \Rightarrow x = z.
\]

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 \texttt{apply} (p. 60) 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{verbatim}
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.
\end{verbatim}

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

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

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

\(R x _ (Q x _)\)

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

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

\subsection{3.3.2 Occurrence Selectors and Rewriting Directions}

Some ambient logic tactics use \textit{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 \textit{not} in the following list. They may also be empty, meaning that all applicable occurrences should be operated on.

Some ambient logic tactics use \textit{rewriting directions}, \texttt{dir}, which may either be empty (meaning rewriting from left to right), or \texttt{-}, meaning rewriting from right to left.

\subsection{3.3.3 Introduction and Generalization}

\textbf{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, \texttt{occ} ranges over occurrence selectors, and \texttt{dir} ranges over directions—see Subsection 3.3.2).

If a list \(i_1, \ldots, i_n\) of introduction patterns consists entirely of \texttt{//} (apply the \texttt{trivial} (p. 55) tactic), \texttt{/=} (apply the \texttt{simplify} (p. 56) tactic) and \texttt{//=} (apply the \texttt{simplify} and then \texttt{trivial}),

\begin{verbatim}
p ::= _ proof hole
(X, q_1, \ldots, q_n) lemma application
q ::= e expression
p proof term
\end{verbatim}

Figure 3.1: Proof Terms
then applying $t_1, \ldots, t_n$ to a list of goals $G_1, \ldots, G_m$ is done by applying the tactics corresponding to the $t_i$ 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.

\[
\tau \Rightarrow t_1 \cdots t_n
\]

Runs the tactic $\tau$, matching the resulting goals, $G_1, \ldots, G_l$, with the introduction patterns $t_1, \ldots, t_n$:

- Suppose $k$ is such that all of $t_1, \ldots, t_{k-1}$ are $//$, $//=\,$, and either $k > n$ or $t_k$ is not $//$, $//=\,$ or $//=\,$.
- Let $G'_1, \ldots, G'_l'$ be the goals resulting from applying $t_1, \ldots, t_{k-1}$ 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 $t_k$ is not a case pattern, each subgoal $G'_i$ is matched against $t_k, \ldots, t_n$ by the procedure described below, with the resulting subgoals being collected into a list of goals (maintaining order viz a viz the indices $i$) as the tactical’s result.
- Otherwise, $t_k$ is a case pattern $[t_{11} \cdots t_{1m_1} | \cdots | t_{r1} \cdots t_{rm_r}]$.
- If $\tau$ is not equivalent to \texttt{idtac} (p. 47), the tactic fails unless $r = l'$, in which case each $G'_i$ is matched against

\[
t_{11} \cdots t_{1m_1}, t_{k+1} \cdots t_n
\]

by the procedure described below, with the resulting subgoals being collected into the tactical’s result.
- Otherwise, $\tau$ is equivalent to \texttt{idtac} (p. 47) (and so $l' = 1$). In this case $G'_1$ 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.

Figure 3.2: Introduction Patterns
Matching a single goal against a list of patterns: To match a goal $G$ against a list of introduction patterns $\iota_1, \ldots, \iota_n$, the introduction patterns are processed from left-to-right, as follows:

- $(b)$ The top assumption (universally quantified identifier, module name or memory; or left side of implication) is consumed, and introduced with this name. Fails if the top assumption has neither of these forms.

- $(b!)$ Same as the preceding case, except that $b$ is used as the base of the introduced name, extending the base to avoid naming conflicts.

- $(\_)$ Same as the preceding case, except the assumption is introduced with an anonymous name (which can’t be uttered by the user).

- $(\+)$ Same as the preceding case, except that after a branch of the procedure completes, yielding a goal, the assumption will be reverted, i.e., un-introduced (using a universal quantifier or implication as appropriate).

- $(\?)$ Same as the preceding case, except EasyCrypt chooses the name by which the assumption is introduced (using universally quantified names as assumption bases).

- $(\text{occ} \to \cdot)$ 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.

- $(\text{occ} \to \cdot)$ The same as the preceding case, except the rewriting is from right-to-left.

- $(\to 
\cdot)$ The same as $\to \cdot$, 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.

- $(\langle \langle)$ The same as the preceding case, except the substitution is from right-to-left.

- $(\text{/}p)$ Replace the top assumption by the result of applying the proof term $p$ to it using forward reasoning.

- $(\{a_1 \cdot \ldots \cdot a_n\})$ 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.

- $(\text{/}p)$ Apply simplify (p. 56) to goal’s conclusion.

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

- $(\text{/}p)$ Apply simplify (p. 56) and then trivial (p. 55) to goal’s conclusion; this may solve the goal, so that the procedure’s current branch yields no goals.

- $(\text{dir occ} \@/\text{op})$ Unfold (fold, if the direction is $-$) the definition of operator $op$ at the specified occurrences of the goal’s conclusion. See the rewrite (p. 62) tactic for the details.

- $(\{0 \text{m}1 \cdot \ldots \cdot 0 \text{m}r \mid \ldots \mid \text{r}1 \cdot \ldots \cdot \text{r}m\})$
  - If $r = 0$, then the top assumption of the goal is destructed using the case (p. 64) tactic, the resulting goals are matched against $t_2, \ldots, t_n$, and their subgoals are assembled into a list of goals.
  - Otherwise $r > 0$. The goal’s top assumption is destructed using the case (p. 64) tactic, yielding subgoals $H_1, \ldots, H_p$. If $p \neq r$, the procedure fails. Otherwise each subgoal $H_i$ is matched against $t_{i1} \cdot \ldots \cdot t_{im_i} t_{i2} \cdot \ldots \cdot t_{in_i}$ with the resulting goals being collected into a list as the procedure’s result.
CHAPTER 3. TACTICS

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
```

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

```
Type variables: <none>

x : int
y : int

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

Then running

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

produces

```
Type variables: <none>

x : int
y : int

forall (z : int), y = z => y = z
```
Alternatively, one can introduce the assumption $x = y$, and then use the $\rightarrow \rightarrow$ substitution pattern: if the current goal is

<table>
<thead>
<tr>
<th>Type variables: &lt;none&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x : \text{int}$</td>
</tr>
<tr>
<td>$y : \text{int}$</td>
</tr>
<tr>
<td>$H1: x = y$</td>
</tr>
<tr>
<td>$z : \text{int}$</td>
</tr>
<tr>
<td>$y = z \Rightarrow x = z$</td>
</tr>
</tbody>
</table>

then running

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

produces

<table>
<thead>
<tr>
<th>Type variables: &lt;none&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x : \text{int}$</td>
</tr>
<tr>
<td>$y : \text{int}$</td>
</tr>
<tr>
<td>$z : \text{int}$</td>
</tr>
<tr>
<td>$H1: x = z$</td>
</tr>
<tr>
<td>$x = z$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Type variables: &lt;none&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Sym: } \forall (u ; v : \text{int}), ; u = v \Rightarrow v = u$</td>
</tr>
<tr>
<td>$x : \text{int}$</td>
</tr>
<tr>
<td>$y : \text{int}$</td>
</tr>
<tr>
<td>$x = y \Rightarrow \forall (z : \text{int}), ; y = z \Rightarrow x = z$</td>
</tr>
</tbody>
</table>

Then running

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

produces

<table>
<thead>
<tr>
<th>Type variables: &lt;none&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Sym: } \forall (u ; v : \text{int}), ; u = v \Rightarrow v = u$</td>
</tr>
<tr>
<td>$x : \text{int}$</td>
</tr>
<tr>
<td>$y : \text{int}$</td>
</tr>
<tr>
<td>$y = x \Rightarrow \forall (z : \text{int}), ; y = z \Rightarrow x = z$</td>
</tr>
</tbody>
</table>

And then running

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

on this goal produces

<table>
<thead>
<tr>
<th>Type variables: &lt;none&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Sym: } \forall (u ; v : \text{int}), ; u = v \Rightarrow v = u$</td>
</tr>
<tr>
<td>$x : \text{int}$</td>
</tr>
<tr>
<td>$y : \text{int}$</td>
</tr>
<tr>
<td>$\forall (z : \text{int}), ; x = z \Rightarrow x = z$</td>
</tr>
</tbody>
</table>
Finally, let’s see examples of how a disjunction assumption may be destructed, either using the `case` tactic followed by a case introduction pattern, or by making the case introduction pattern do the destruction. For the first case, if the current goal is

```
Type variables: <none>

x : bool
y : bool

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

then running

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

produces the two goals

```
Type variables: <none>

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

And for the second case, if the current goal is

```
Type variables: <none>

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

then running

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

produces the two goals

```
Type variables: <none>

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

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

\[ \tau : \pi_1 \ldots \pi_n \]

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. 64) (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
  
  y = x
  
  then running
  
  move: x eq_xy.

  produces

  Type variables: <none>

  y : int

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

  In this example, one can’t generalize \(x\) without also generalizing \(eq_xy\).

- \(\pi\) 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
CHAPTER 3. TACTICS

Type variables: <none>

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

then running

\[
\text{move: } (y = x).
\]

produces

Type variables: <none>

\[
\begin{align*}
x &: \text{int} \\
y &: \text{int} \\
\forall (x_0 : \text{bool}), x = y \implies x_0
\end{align*}
\]

Alternatively, running

\[
\text{move: } \{2\} y \{2\} x.
\]

produces

Type variables: <none>

\[
\begin{align*}
x &: \text{int} \\
y &: \text{int} \\
\forall (y_0 x_0 : \text{int}), x = y \implies y_0 = x_0
\end{align*}
\]

3.3.4 Tactics

\begin{itemize}
\item \texttt{idtac}
  
  Does nothing, i.e., leaves the goal unchanged.
\item \texttt{move}
  
  Does nothing, equivalent to \texttt{idtac} (p. 47). It is mainly used in conjunction with the introduction tactical and the generalization mechanism. See Section 3.3.3.
\item \texttt{clear} \(a_1 \cdots a_n\)
  
  Clear assumptions \(a_1 \cdots a_n\) from the goal’s context. Fail if any remaining hypotheses depend on any of the \(a_i\).
  
  For example, if the current goal is

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

  then running


**clear** z eq_yz.

produces

Type variables: <none>

x : int
y : int
eq_xy: x = y

x - 1 = y - 1

**assumption**

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

For example, if the current goal is

Type variables: <none>

x : int
y : int
eq_xy: x = y

x = y

then running

**assumption.**

solves the goal.

**reflexivity**

Solve goals with conclusions of the form \( b = b \) (up to computation).

For example, if the current goal is

Type variables: <none>

y : bool

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

then running

**reflexivity.**

solves the goal.

**left**

Reduce a goal whose conclusion is a disjunction to one whose conclusion is its left member.

For example, if the current goal is

Type variables: <none>

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

x = y + 1 \lor y = x + 1

then running
produces the goal

```
left.
```

Type variables: <none>

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

```
x = y + 1
```

\[\text{right}\]

Reduce a goal whose conclusion is a disjunction to one whose conclusion is its right member.

For example, if the current goal is

```
Type variables: <none>

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

```
x = y + 1 \lor y = x + 1
```

then running

```
right.
```

produces the goal

```
Type variables: <none>

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

```
y = x + 1
```

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

```
Type variables: <none>

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

```
x = y + 1 || y = x + 1
```

then running

```
right.
```

produces the goal

```
Type variables: <none>

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

```
x <> y + 1 \Rightarrow y = x + 1
```

3. TACTICS

● **exists e**

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

```
Type variables: <none>

x : int
rng_x: 0 < x < 5

exists (x0 : int), 5 < x0 < 10
```

then running

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

produces the goal

```
Type variables: <none>

x : int
rng_x: 0 < x < 5

5 < x + 5 < 10
```

● **split**

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

- close any goal that is convertible to true or provable by reflexivity,
- replace a logical equivalence by the direct and indirect implications,
- replace a goal of the form $\phi_1 \land \phi_2$ 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

```
Type variables: <none>

x : int
y : int

x = y ↔ x - 1 = y - 1
```

then running

```
split.
```

produces the goals

```
Type variables: <none>

x : int
y : int

x = y => x - 1 = y - 1
```

and
CHAPTER 3. TACTICS

Type variables: <none>

\[ x : \text{int} \]
\[ y : \text{int} \]

\[ x - 1 = y - 1 \Rightarrow x = y \]

And if the current goal is

Type variables: <none>

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

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

then running

\textit{split}.

produces the goals

Type variables: <none>

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

\[ x - 1 = y - 1 \]

and

Type variables: <none>

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

\[ z + 1 = w + 1 \]

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

Type variables: <none>

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

\[ x - 1 = y - 1 \land z + 1 = w + 1 \]
then running

\texttt{split}.

produces the goals

\begin{verbatim}
Type variables: <none>

x : int
y : int
z : int
w : int
eq_xy: x = y
eq_zw: z = w

\hline
x - 1 = y - 1
\end{verbatim}

and

\begin{verbatim}
Type variables: <none>

x : int
y : int
z : int
w : int
eq_xy: x = y
eq_zw: z = w

\hline
x - 1 = y - 1 => z + 1 = w + 1
\end{verbatim}

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

\begin{verbatim}
Type variables: <none>

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

\hline
(x - 7, 2 + y) = (z + 2, w - 10)
\end{verbatim}

then running

\texttt{split}.

produces the goals

\begin{verbatim}
Type variables: <none>

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

\hline
x - 7 = z + 2
\end{verbatim}

and
Type variables: <none>

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

2 + y = w - 10

\[\circ \text{congr}\]

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

Type variables: <none>

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

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

then running

\[\text{congr}\]

produces the goals

Type variables: <none>

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

\[x + 1 = x' - 1\]

and

Type variables: <none>

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

\[y - 1 = y' + 1\]
And if the current goal is

```plaintext
Type variables: <none>

x : int
y : int
x' : int
y' : int
eq_xx' : x = x' - 2
eq_yy' : y = y' + 2

x + 1 + (y - 1) = x' - 1 + (y' + 1)
```

then running

```
congr.
```

produces this same pair of subgoals.

```
subst x | subst
```

**Syntax:** `subst x`. Search for the first equation of the form \( x = t \) or \( t = x \) in the context and replace all the occurrences of \( x \) by \( t \) everywhere in the context and the conclusion before clearing it.

For example, if the current goal is

```plaintext
Type variables: <none>

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

x = w
```

then running

```
subst x.
```

takes us to

```plaintext
Type variables: <none>

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

y = w
```

from which running

```
subst y.
```

takes us to

```plaintext
Type variables: <none>

z : bool
w : bool
```
CHAPTER 3. TACTICS

\begin{verbatim}
eq_zw: z = w

\hline
\end{verbatim}
from which running

\begin{verbatim}
subst z.

\hline
\end{verbatim}
takes us to

\begin{verbatim}
Type variables: <none>

w : bool

w = w

\hline
\end{verbatim}

\textbf{Syntax:} \texttt{subst}. Repeatedly apply \texttt{subst} \textit{x} to all identifiers in the context.

For example, if the current goal is

\begin{verbatim}
Type variables: <none>

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

\hline
\end{verbatim}
then running

\begin{verbatim}
subst.

\hline
\end{verbatim}
takes us to

\begin{verbatim}
Type variables: <none>

x : bool

x = x

\hline
\end{verbatim}

\texttt{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

\begin{verbatim}
Type variables: <none>

\hline
\end{verbatim}

\begin{verbatim}
forall (x y : int), x = y \Rightarrow y - 1 = x - 1

\hline
\end{verbatim}
then running

\begin{verbatim}
trivial.

\hline
\end{verbatim}
solves the goal. On the other hand, if the current goal is

\begin{verbatim}
Type variables: <none>

\hline
\end{verbatim}

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

\hline
\end{verbatim}
then running

\texttt{trivial.}

leaves the goal unchanged.

\textbf{Fixme} Note: Be a bit more detailed about what this tactic does?

② \textbf{done}

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

② \textbf{simplify} \mid \texttt{simplify} x_1 \cdots x_n \mid \texttt{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 \texttt{delta} is given. The $\delta$-reduction can be restricted to a set of defined symbols by replacing \texttt{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 \texttt{beta} (resp. \texttt{iota}, \texttt{zeta}, \texttt{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

\begin{verbatim}
Type variables: <none>

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

true => true <=\ (y => true \& z)
\end{verbatim}

then running

\texttt{simplify.}

produces the goal

\begin{verbatim}
Type variables: <none>

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

y => z
\end{verbatim}

And if the current goal is

\begin{verbatim}
Type variables: <none>

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

false => false <=\ (y => false \& z)
\end{verbatim}

then running

\texttt{simplify.}

produces the goal

\begin{verbatim}
Type variables: <none>

y : bool
z : bool

y : bool
z : bool
CHAPTER 3. TACTICS

imp_yz: y => z
true

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

\textbf{progress} | \textbf{progress} \tau

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

For example, if the current goal is

\begin{verbatim}
forall (x y z : bool), (x \land y => z) <=> (x => y => z)
\end{verbatim}

then running

\textbf{progress}.

\textbf{Fixme} Note: Describe \textbf{progress} options.

\textbf{smt} smt-options

Try to solve the goal using SMT solvers. The goal is sent along with the local hypotheses plus selected axioms and lemmas. The SMT solvers used, their options, and the axiom selection algorithm are specified by \textit{smt-option}.

For example, if the current goal is

\begin{verbatim}
forall (x y z : bool), (x \land y => z) <=> (x => y => z)
\end{verbatim}

then running

\textbf{smt}.

solves the goal.

\textbf{Fixme} Note: Describe failure states of prover selection.

\textbf{Options}

- \textit{timeout=} \(n\): set the timeout for provers to \(n\) (in seconds).
- \textit{maxprovers=} \(n\): set the maximum number of provers running in parallel to \(n\)
- \textit{prover=} \((\text{prover-selector})\): select the provers, where \textit{prover-selector} is a list of modified prover names:
  - `"prover-name": use the listed prover;
  - `+"prover-name": add \textit{prover-name} to the current list of provers;
  - `-"prover-name": remove \textit{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;

\textbf{Fixme} Note: Describe failure states of prover selection.

- Axiom selection: axioms and lemmas are not all send to smt provers, EASYCRYPT use a strategy to automatically select them. Lemmas and axioms marked with “nosmt” are not selected by default. This strategy can be parametrized using different options:
  - unwantedlemmas=\textit{dbhint}: do not send axiom/lemma selected by \textit{dbhint}
  - wantedlemmas=\textit{dbhint}: send axiom/lemma selected by \textit{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:

\texttt{smt 100 [+''Z3] tmo=4 mp=2}

is equivalent to

\texttt{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: \texttt{prover smt-options} Fixme Note: Make this a pragma?

Remark: By default, smt failures cannot be caught by the try (p. 69) tactical.

\begin{itemize}
  \item \texttt{admit}
  \begin{itemize}
    \item Close the current goal by admitting it.
    \item For example, if the current goal is
    \begin{Verbatim}
    Type variables: <none>
    x : int
    y : int
    x = y + 1 \land x = y - 1
    \end{Verbatim}
    \end{itemize}
  \end{itemize}

then running
\begin{Verbatim}
admit.
\end{Verbatim}

solves the goal.

\begin{itemize}
  \item \texttt{change φ}
  \begin{itemize}
    \item Replace the current goal’s conclusion by $φ \land φ$ must be convertible to the current goal’s conclusion.
    \item For example, if the current goal is
    \begin{Verbatim}
    Type variables: <none>
    y : bool
    z : bool
    imp_yz: y => z
    true => true <=> (y => true /\ z)
    \end{Verbatim}
    \end{itemize}
  \end{itemize}

then running
\begin{Verbatim}
change (y => z).
\end{Verbatim}

produces the goal
\begin{Verbatim}
Type variables: <none>
  y : bool
  z : bool
\end{Verbatim}
CHAPTER 3. TACTICS

imp_yz: y => z

y => z

pose x := π

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

For example, if the current goal is

```
Type variables: <none>
x : int
y : int
2 * ((x + 1 + y) * x) = (x + 1 + y) * x + (x + 1 + y) * x
```

then running

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

produces the goal

```
Type variables: <none>
x : int
y : int
z : int := (x + 1 + y) * x
2 * z = z + z
```

have \( ι : φ \)

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

For example, if the current goal is

```
Type variables: <none>
x : bool
notnot_x: (x => false) => false
```

then running

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

produces the goals

```
Type variables: <none>
x : bool
notnot_x: (x => false) => false
```

\[ x \lor (x => false) \]

and
Type variables: <none>

\texttt{x : bool}
\texttt{notnot\_x: (x \Rightarrow \text{false}) \Rightarrow \text{false}}
\texttt{excl\_or: x \lor/\ (x \Rightarrow \text{false})}
\texttt{x}

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

\texttt{cut }\iota:\phi

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

\texttt{apply }p | \texttt{apply }/p_1 \cdots /p_n | \texttt{apply }p \texttt{ in }H

\textbf{Syntax:} \texttt{apply }p. \textit{Tries to match the conclusion of the proof term }p\textit{ 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}

\begin{verbatim}
pred P : int.
pred Q : int.
pred R : int.
axiom P (x : int) : P x.
axiom Q (x : int) : P x \Rightarrow Q x.
axiom R (x : int) : P(x + 1) \Rightarrow Q x \Rightarrow R x.
\end{verbatim}

If the current goal is

Type variables: <none>

\texttt{x : int}
\texttt{R x}

then running

\texttt{apply }R.

produces the goals

Type variables: <none>

\texttt{x : int}
\texttt{P (x + 1)}

and

Type variables: <none>

\texttt{x : int}
\texttt{Q x}

And running

\texttt{apply }\texttt{(R x \ _ \ (Q x \ _))}.

from that initial goal produces the goals
CHAPTER 3. TACTICS

Type variables: <none>

\[ x : \text{int} \]

\[ P(x + 1) \]

and

Type variables: <none>

\[ x : \text{int} \]

\[ P x \]

Syntax: \texttt{apply} /\!\!p_1 \cdots /\!\!p_n.\texttt{ Apply the proof terms }p_1,\ldots, p_n\texttt{ 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\texttt{, we have whatever goals } p_1\texttt{ has produced from the current goal. } p_2\texttt{ is applied to the last of these goals, and that last goal is replaced by the goals produced by running } p_2\texttt{, etc. Fails without changing the goal if any of these applications fails. For example, if the current goal is}

Type variables: <none>

\[ x : \text{int} \]

\[ R x \]

then running

\texttt{apply} /\!\!R /\!\!Q /\!\!P /\!\!P.\texttt{ solves the goal.}

Syntax: \texttt{apply} \!\!p \texttt{ in } H. \texttt{ Apply } p \texttt{ in forward reasoning to } H, \texttt{ replacing } H \texttt{ by the result. For example, if the current goal is}

Type variables: <none>

\[ x : \text{int} \]

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

\[ Q \: x \implies R \: x \]

then running

\texttt{apply} R \texttt{ in } HP.\texttt{ produces the goal}

Type variables: <none>

\[ x : \text{int} \]

\[ \text{HP: } Q \: x \implies R \: x \]

\[ Q \: x \implies R \: x \]

\[ \triangleright \]

Syntax: \texttt{exact} \!\!p | \texttt{exact} /\!\!p_1 \cdots /\!\!p_n. \texttt{ Equivalent to } \texttt{by apply} \!\!p, \texttt{ i.e., apply the given proof-term and then try to close the goals with } \texttt{trivial} —failing if not all goals can be closed.

Syntax: \texttt{exact} /\!\!p_1 \cdots /\!\!p_n. \texttt{ Equivalent to } \texttt{by apply} /\!\!p_1 \cdots /\!\!p_n.
CHAPTER 3. TACTICS

\[ rewrite \pi_1 \cdots \pi_n \ | \ rewrite \pi_1 \cdots \pi_n \ in H \]

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

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

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

Depending on the form of \( \pi \), \( rewrite \pi \) does the following:

- For //, /=, and //=, see Subsection 3.3.3.
- If \( \pi \) is a proof-term with conclusion \( f_1 = f_2 \), then \( rewrite \) searches for the first subterm of the goal’s conclusion matching \( f_1 \) and resulting in the full instantiation of the pattern. It then replaces, after instantiation of the pattern, all the occurrences of \( f_1 \) by \( f_2 \) in the goal’s conclusion, and creates new subgoals for the instantiations of the 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 \( f_1 <=> f_2 \). If the direction indicator - is given, \( rewrite \) works in the reverse direction, searching for a match of \( f_2 \) and then replacing all occurrences of \( f_2 \) by \( f_1 \).
- If \( \pi \) is a /-prefixed pattern of the form \( op_1 \cdots p_n \), with \( o \) a defined symbol, then \( rewrite \) searches for the first subterm of the goal’s conclusion matching \( op_1 \cdots p_n \) and resulting in the full instantiation of the pattern. It then replaces, after instantiation of the pattern, all the occurrences of \( op_1 \cdots p_n \) by the \( \beta \delta \) head-normal form of \( op_1 \cdots p_n \), where the \( \delta \)-reduction is restricted to subterms headed by the symbol \( o \). If no subterms of the goal’s conclusion match \( op_1 \cdots p_n \) or if the pattern cannot be fully instantiated by matching, the tactic fails. If the direction indicator - is given, \( rewrite \) works in the reverse direction, searching for a match of the \( \beta \delta \) head-normal of \( op_1 \cdots p_n \) and then replacing all occurrences of this head-normal form with \( op_1 \cdots p_n \).

The occurrence selector restricts which occurrences of the matching pattern are replaced in the goal’s conclusion—see Subsection 3.3.3.

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

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

For example, if the current goal is

\[
\begin{align*}
\text{Type variables:} & \ <\text{none}> \\
x : & \ \text{int} \\
y : & \ \text{int} \\
eq_{xy} : & \ x = y \\
\forall (z : \text{int}), & \ y = z => x = z
\end{align*}
\]

then running

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

produces
CHAPTER 3. TACTICS

Type variables: <none>

x : int
y : int
eq_xy: x = y

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

from which running

rewrite - {1} eq_xy.

produces

Type variables: <none>

x : int
y : int
eq_xy: x = y

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

from which running

rewrite - eq_xy.

produces

Type variables: <none>

x : int
y : int
eq_xy: x = y

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

Syntax: rewrite π₁ · · · πₙ in H. Like the preceding case, except rewriting is done in the hypothesis H instead of in the goal’s conclusion. Rewriting using a proof term is only allowed when the proof term was defined globally or before the assumption H.

For example, if the current goal is

Type variables: <none>

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

x = z

then running

rewrite - eq_xy in eq_xz.

produces

Type variables: <none>

x : int
y : int
eq_xy: x = y
CHAPTER 3. TACTICS

\[
z : \text{int} \\
eq_{xz} : x = z
\]

\[
x = z
\]

⚠️ **case φ | case**

**Syntax:** case φ. Assuming the goal’s conclusion is *not* a statement judgement, do an excluded-middle case analysis on φ, substituting φ in the goal’s conclusion.

For example, if the current goal is

```plaintext
Type variables: <none>

x : int
y : int
abs_bnd: \(|x - y| \leq 10

x - y \leq 10 \lor y - x \leq 10
```

then running

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

produces the goals

```plaintext
Type variables: <none>

x : int
y : int
abs_bnd: \(|x - y| \leq 10

x <= y => x - y <= 10 \lor y - x <= 10
```

and

```plaintext
Type variables: <none>

x : int
y : int
abs_bnd: \(|x - y| \leq 10

! x <= y => x - y <= 10 \lor y - x <= 10
```

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

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

```plaintext
Type variables: <none>

x : int
y : int
z : int

x = y \lor y = z => x = z
```

then running

```
case.
```
produces the goal

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

\&\& works identically.

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

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

then running

\[
\text{case.}
\]

produces the goals

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

and

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

| | works identically.

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

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

then running

\[
\text{case.}
\]

produces the goal

\[
\begin{align*}
\text{Type variables: & <none>} \\
\text{x : int}\\
\end{align*}
\]
\[
\begin{align*}
forall \ (x : \text{int}), \ 0 < x < 5 \Rightarrow exists \ (x0 : \text{int}), \ 5 < x0 < 10
\end{align*}
\]
• (unit) Substitutes \( \texttt{tt} \) for the assumption in the remainder of the conclusion.

• (bool) For example, if the current goal is

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

\[
\text{forall } (x \ y : \text{bool}), \ x \leftrightarrow y \Rightarrow x = y
\]

then running

\[
\text{case.}
\]

produces the goals

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

\[
\text{forall } (y : \text{bool}), \ \text{true} \leftrightarrow y \Rightarrow \text{true} = y
\]

and

\[
\text{forall } (y : \text{bool}), \ \text{false} \leftrightarrow y \Rightarrow \text{false} = y
\]

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

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

\[
\text{forall } (x \ y : \text{bool} \times \text{bool}), \ x \neq y \Rightarrow x._1 \neq y._1 \lor x._2 \neq y._2
\]

then running

\[
\text{case.}
\]

produces the goal

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

\[
\text{forall } (x_1 \ x_2 : \text{bool} \times \text{bool}), \ (y : \text{bool} \times \text{bool}),
\quad
(x_1, x_2) \neq y \Rightarrow
\quad
(x_1, x_2)._1 \neq y._1 \lor (x_1, x_2)._2 \neq y._2
\]

• (inductive datatype) Consider the inductive datatype declaration:

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

Then, if the current goal is

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

\[
\text{forall } (\text{tr} : \text{tree}),
\quad
\text{tr} = \text{Leaf} \lor
\quad
\exists (b : \text{bool}) \ (\text{tr}_1 \text{tr}_2 : \text{tree}), \ \text{tr} = \text{Node} \ b \ \text{tr}_1 \text{tr}_2
\]
then running

```latex
\texttt{case.}
```

produces the goals

Type variables: <none>

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

and

Type variables: <none>

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

© \texttt{elim | elim /L}

**Syntax:** \texttt{elim}. Eliminates the top assumption of the goal's conclusion, generating subgoals that are dependent upon the kind of assumption eliminated. *This tactic can be followed by the generalization tactical—see Subsection 3.3.3.*

\texttt{elim} mostly works identically to \texttt{case} (p. 64), the exception being inductive datatype and the integers (for which a built-in induction principle is applied—see the other form).

Consider the inductive datatype declaration:

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

Then, if the current goal is

Type variables: <none>

```
p : tree \rightarrow \texttt{bool}
basis: p Leaf
indstep: \texttt{forall (b : bool) (tr1 tr2 : tree),}
\hspace{1em} p tr1 \Rightarrow p tr2 \Rightarrow p (Node b tr1 tr2)
```

running

```
\texttt{elim.}
```

produces the goals

Type variables: <none>

```
p : tree \rightarrow \texttt{bool}
basis: p Leaf
indstep: \texttt{forall (b : bool) (tr1 tr2 : tree),}
\hspace{1em} p tr1 \Rightarrow p tr2 \Rightarrow p (Node b tr1 tr2)
```

```
p Leaf
```

and
Type variables: <none>

\begin{verbatim}
p : tree -> bool
basis: p Leaf
indstep: forall (b : bool) (tr1 tr2 : tree),
  p tr1 => p tr2 => p (Node b tr1 tr2)
forall (b : bool) (t t0 : tree), p t => p t0 => p (Node b t t0)
\end{verbatim}

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

\begin{verbatim}
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).
\end{verbatim}

and suppose we’ve already proved

\begin{verbatim}
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.
\end{verbatim}

Then, if the current goal is

\begin{verbatim}
forall (t : tree), rev (rev t) = t
\end{verbatim}

running

\begin{verbatim}
elim /IndPrin.
\end{verbatim}

produces the goals

\begin{verbatim}
forall (t : tree), rev (rev t) = t
\end{verbatim}

and

\begin{verbatim}
forall (b : bool) (tr1 tr2 : tree),
  rev (rev tr1) = tr1 =>
  rev (rev tr2) = tr2 => rev (rev (Node b tr1 tr2)) = Node b tr1 tr2
\end{verbatim}

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

\section*{algebra}

\textbf{Fixme} Note: Missing description of \texttt{algebra}. 
3.3.5 Tacticals

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

\[ \tau_1; \tau_2 \]

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 \& y) \& z \Rightarrow x \& y \& z
```

then running

```
case; case.
```

produces the goals

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

x \Rightarrow y \Rightarrow z \Rightarrow x \& y \& z
```

\[ \tau_1: [\tau_1 \mid \cdots \mid \tau_n] \]

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 \& y \& z
```

then running

```
split; [assumption | split; 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.

\( \odot \) \texttt{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 it \( \tau \) does not apply to the current goal.

For example, if the current goal is

\begin{verbatim}
Type variables: <none>
x : bool
y : bool
z : bool
w : bool

((x /
  y) /
  z) /
  w /
  (x /
  y /
  z) /
  w => w
\end{verbatim}

then running

\begin{verbatim}
   do! case.
\end{verbatim}

produces the goals

\begin{verbatim}
Type variables: <none>
x : bool
y : bool
z : bool
w : bool

x => y => z => w => w
\end{verbatim}

and

\begin{verbatim}
Type variables: <none>
x : bool
y : bool
z : bool
w : bool

x => y /
  z => w => w
\end{verbatim}

**Variants.**

\begin{description}
\item \texttt{do? }\tau \quad \text{apply}\ \tau\ 0\ or\ more\ times,\ until\ it\ fails
\item \texttt{do n! }\tau \quad \text{apply}\ \tau\ with\ depth\ exactly\ n
\item \texttt{do n? }\tau \quad \text{apply}\ \tau\ with\ depth\ at\ most\ n
\end{description}

\( \odot \) \texttt{\tau; 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 \( \tau_1 \)

\( \tau; \text{first } n \tau \) reorder the subgoals generated by \( \tau \), moving the first \( n \) to the end of the list

\( \tau; \text{last } n \tau \) reorder the subgoals generated by \( \tau \), moving the last \( n \) to the beginning of the list

\( \tau; \text{last first } \tau \) reorder the subgoals generated by \( \tau \), moving the last one to the beginning of the list

\( \tau; \text{first last } \tau \) reorder the subgoals generated by \( \tau \), moving the first one to the end of the list

For example, if the current goal is

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

then running

\[
\text{split; last first.}
\]

produces the goals

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

and

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

Apply the tactic \( \tau \) and try to close all the generated subgoals using \texttt{trivial} (p. 55). Fail if not all subgoals can be closed.

Remark. Inside the a lemma’s proof, \texttt{by []} is equivalent to \texttt{by trivial}. But the form

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

means

\[
\text{lemma} \ldots \texttt{by (trivial; smt)}.
\]
3.4 Program Logics

In this section, we describe the tactics of EASYCRYPT’s three program logics: \textit{pRHL}, \textit{pHL} and \textit{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 \textit{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.

### 3.4.1 Tactics for Reasoning about Programs

\textbf{proc}

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

For example, if the current goal is

\begin{verbatim}
Type variables: <none>

pre = ={x, y}
G1.f ~ G2.f
post = ={res}
\end{verbatim}

then running

\begin{verbatim}
proc.
\end{verbatim}

produces the goal

\begin{verbatim}
Type variables: <none>

&1 (left ) : G1.f
&2 (right ) : G2.f
pre =
(x{1}, y{1}).'1 = (x{2}, y{2}).'1 /
(x{1}, y{1}).'2 = (x{2}, y{2}).'2
x <- x + 1 (1) y <- y + 1
y <- y + 1 (2) x <- x + 1
post = x{1} + y{1} = x{2} + y{2}
\end{verbatim}
CHAPTER 3. TACTICS

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

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

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

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

if the current goal is

```verbatim
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

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

produces the goals

```verbatim
Type variables: <none>
M : T{Or}
forall \&1 \&2,
```
The tactic would fail without the module restriction $T\{\text{Or}\}$ on $M$, as then $M$ could directly manipulate $\text{Or}.x$. It would also fail if, in the declaration of the module type $T$, $g$ were given access to $\text{Or}.f_3$.

**Syntax:** $\text{proc } B \ I$. Like $\text{proc } I$, 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

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

op low : int = -100.
op upp : int = 100.
```
module Or1 : OR = {
  var qry, rsp : int
  var queried : bool

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

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

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

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

Then, if the current goal is

Type variables: <none>

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

pre = Or1.qry{1} = Or2.qry{2}
Adv(Or1).f ~ Adv(Or2).f
post = !Or2.queried{2} => ={res}

running

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

produces the goals

Type variables: <none>

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

forall &1 &2, 
  Or1.qry{1} = Or2.qry{2} =>
  !Or2.queried{2} => true /\ Or1.qry{1} = Or2.qry{2}
and

Type variables: <none>

Adv: ADV{Or1, Or2}
\( l_1 \text{Adv}_f: \forall (O <: OR\{Adv\}), \)
\( \text{islossless} \ O.qry \Rightarrow \text{islossless} \ Adv(O).f \)

\( \forall &1, &2, \)
\( \neg \text{Or2.queried}(2) \Rightarrow \)
\( \{\text{res}\} \ \land \ \{\text{glob Adv} \ \land \ Or1.qry(1) = Or2.qry(2)\} \Rightarrow \)
\( \neg \text{Or2.queried}(2) \Rightarrow \{\text{res}\} \)

and

Type variables: <none>

Adv: ADV{Or1, Or2}
\( l_1 \text{Adv}_f: \forall (O <: OR\{Adv\}), \)
\( \text{islossless} \ O.qry \Rightarrow \text{islossless} \ Adv(O).f \)

\( \forall (O <: OR\{Adv\}), \text{islossless} \ O.qry \Rightarrow \text{islossless} \ Adv(O).f \)

pre = \( \neg \text{Or2.queried}(2) \ \land \ \{\text{x}\} \ \land \ Or1.qry(1) = Or2.qry(2) \)
\( Or1.qry \land \neg Or2.qry \)

post = \( \neg \text{Or2.queried}(2) \Rightarrow \{\text{res}\} \ \land \ Or1.qry(1) = Or2.qry(2) \)

and

Type variables: <none>

Adv: ADV{Or1, Or2}
\( l_1 \text{Adv}_f: \forall (O <: OR\{Adv\}), \)
\( \text{islossless} \ O.qry \Rightarrow \text{islossless} \ Adv(O).f \)

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

and

Type variables: <none>

Adv: ADV{Or1, Or2}
\( l_1 \text{Adv}_f: \forall (O <: OR\{Adv\}), \)
\( \text{islossless} \ O.qry \Rightarrow \text{islossless} \ Adv(O).f \)

\( \forall _, \text{phoare} \[ \text{Or2.qry} : \text{Or2.queried} \ \land \ \text{true} \Rightarrow \text{Or2.queried} \ \land \ \text{true} \] = 1%r \)

Syntax: \textit{proc} \( B \ I \ J \). Like \textit{proc} \( B \ I \), but where the extra invariant, \( J \), holds after failure has occurred. In the \textit{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} \ O1.queried{1} = Or2.queried{2}
```

```
Adv(Or1).f - Adv(Or2).f
```

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

then running

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

produces the goals

```
Type variables: <none>

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

```
forall &1 &2,
    Or1.qry{1} = Or2.qry{2} \ Or1.queried{1} = Or2.queried{2} =>
    if Or2.queried{2} then Or1.queried{1} = Or2.queried{2}
    else
        true \ /
        Or1.qry{1} = Or2.qry{2} \ Or1.queried{1} = Or2.queried{2}
```

and

```
Type variables: <none>

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

```
forall &1 &2,
    if Or2.queried{2} then Or1.queried{1} = Or2.queried{2}
    else
        ={res} \ /
        ={glob Adv} \ /
        Or1.qry{1} = Or2.qry{2} \ Or1.queried{1} = Or2.queried{2} =>
        Or1.queried{1} = Or2.queried{2} \ (!Or2.queried{2} => ={res})
```

and

```
Type variables: <none>

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

and

Type variables: <none>

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

pre =
!Or2.queried{2} /
={x} /
Or1.qry{1} = Or2.qry{2} /
Or1.queried{1} = Or2.queried{2}

Or1.qry - Or2.qry

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

and

Type variables: <none>

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

forall &2,
Or2.queried{2} =>
phoare[ Or1.qry :
  Or1.queried = Or2.queried{2} =>
  Or1.queried = Or2.queried{2} ] = 1%r

and

Type variables: <none>

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

forall &1,
phoare[ Or2.qry :
  Or2.queried /\ Or1.queried{1} = Or2.queried =>
  Or2.queried /\ Or1.queried{1} = Or2.queried ] = 1%r

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

For example, if the current goal is

Type variables: <none>

pre = ={x, y}

G1.f ~ G2.f
post = \{{\text{res}}\}

then running

\texttt{proc*}.

produces the goal

Type variables: <none>

\&1 (left) : G1.f
\&2 (right) : G2.f
pre = \{(x, y)\}
\(r <@ G1.f(x, y) \text{ (1) } r <@ G2.f(x, y)\)
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.

\textbf{\texttt{skip}}

If the goal’s conclusion is a statement judgement whose program(s) are empty, reduce it to the goal whose conclusion is the ambient logic formula \(P \implies Q\), 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) \land \{y\}\}
post = x(1) + y(1) = -(x(2) - y(2))

then running

\texttt{skip}.

produces the goal

Type variables: <none>

\texttt{forall} \&1 \&2, x(1) = -x(2) \land \{y\} \rightarrow x(1) + y(1) = -(x(2) - y(2))

\textbf{\texttt{seq}}

Syntax: \texttt{seq} \(n_1\ n_2 : R\). **PRHL sequence rule.** If \(n_1\) and \(n_2\) are natural numbers and the goal’s conclusion is a PRHL statement judgement with precondition \(P\), postcondition \(Q\) and such that the lengths of the first and second programs are at least \(n_1\) and \(n_2\), respectively, then reduce the goal to two subgoals:
• A first goal whose conclusion has precondition \( P \), postcondition \( R \), first program consisting of the first \( n_1 \) statements of the original goal’s first program, and second program consisting of the first \( n_2 \) statements of the original goal’s second program.

• A second goal whose conclusion has precondition \( R \), postcondition \( Q \), first program consisting of all but the first \( n_1 \) statements of the original goal’s first program, and second program consisting of all but the first \( n_2 \) statements of the original goal’s second program.

For example, if the current goal is

```
Type variables: <none>

&1 (left) : G1.f
&2 (right) : G2.f
pre = ={x, y}
x <- x + 1 (1) y <- y + 1
y <- y + 1 (2) x <- x + 1
post = x{1} + y{1} = x{2} + y{2}
```

then running

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

produces the goals

```
Type variables: <none>

&1 (left) : G1.f
&2 (right) : G2.f
pre = ={x, y}
x <- x + 1 (1) y <- y + 1
post = x{1} = x{2} + 1 /\ y{2} = y{1} + 1
```

and

```
Type variables: <none>

&1 (left) : G1.f
&2 (right) : G2.f
pre = x{1} = x{2} + 1 /\ y{2} = y{1} + 1
y <- y + 1 (1) x <- x + 1
post = x{1} + y{1} = x{2} + y{2}
```

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

• A first goal whose conclusion has precondition \( P \), postcondition \( R \), and program consisting of the first \( n \) statements of the original goal’s program.
• A second goal whose conclusion has precondition $R$, postcondition $Q$, and program consisting of all but the first $n$ statements of the original goal's program.

For example, if the current goal is

```
Type variables: <none>

Context : M.f
pre = x \%\% 2 = 0
(1) x \leftarrow x + 1
(2) x \leftarrow x + 1
post = x \%\% 2 = 0
```

then running

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

produces the goals

```
Type variables: <none>

Context : M.f
pre = x \%\% 2 = 0
(1) x \leftarrow x + 1
post = x \%\% 2 = 1
```

and

```
Type variables: <none>

Context : M.f
pre = x \%\% 2 = 1
(1) x \leftarrow x + 1
post = x \%\% 2 = 0
```

\[\text{Syntax:}\ sp\] If the goal's conclusion is a \texttt{pRHL}, \texttt{pHL} or \texttt{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 \texttt{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>
```

```
Context : M.f
pre = x \%\% 2 = 0
(1) x \leftarrow x + 1
post = x \%\% 2 = 1
```

and

```
Context : M.f
pre = x \%\% 2 = 1
(1) x \leftarrow x + 1
post = x \%\% 2 = 0
```
&1 (left) : M.f  
&2 (right) : N.f

pre = ={y}

\[
\begin{align*}
  \text{if } (y = 0) \{ & \text{(1--)} \hspace{1em} \text{if } (y = 0) \{ \\
  & y \leftarrow 2 \hspace{1em} \text{(1.1)} \hspace{1em} y \leftarrow 3 \\
  \} \hspace{1em} \text{else } \{ & \text{(1--) } \text{else } \{ \\
  & y \leftarrow y - 2 \hspace{1em} \text{(1?1)} \hspace{1em} y \leftarrow y - 1 \\
  \} \hspace{1em} \}, \\
  x <@ M.g() \hspace{1em} \text{(2--)} \hspace{1em} x <$ [1..10]
\end{align*}
\]

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

then running

\[sp.\]

produces the goal

Type variables: <none>

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

pre =

\[
\begin{align*}
  \exists (y_R : \text{int}), \\
  y(2) = 3 \land \\
  y_R = 0 \land \\
  ((\exists (y_L : \text{int}), y(1) = 2 \land y_L = y_R) \lor \\
  \exists (y_L : \text{int}), y(1) = y_L - 2 \mid y_L \neq 0 \land y_L = y_R) \lor \\
  \exists (y_R : \text{int}), \\
  y(2) = y_R - 1 \land \\
  y_R \leftarrow 0 \land \\
  ((\exists (y_L : \text{int}), y(1) = 2 \land y_L = y_R) \lor \\
  \exists (y_L : \text{int}), y(1) = y_L - 2 \mid y_L \neq 0 \land y_L = y_R)
\end{align*}
\]

\[
\begin{align*}
  x <@ M.g() \hspace{1em} \text{(1)} \hspace{1em} x <$> [1..10]
\end{align*}
\]

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

Syntax: sp n₁ n₂. In pRHL, let sp consume exactly n₁ statements of the first program and n₂ statements of the second program. Fails if this isn’t possible.

Syntax: sp n. In pHL and HL, let sp consume exactly n statements of the program. Fails if this isn’t possible.

.syntax wp.

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

For example, if the current goal is

Type variables: <none>
\[ \&1 \text{ (left) : M.f} \]
\[ \&2 \text{ (right) : N.f} \]

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

\[
x \ll M.g() \quad (1--) \quad x \ll [1..10] 
\]

\[
\text{if (y = 0) \{ } \quad (2--) \quad \text{if (y = 0) \{ } \\
\quad y \leftarrow 2 \quad (2.1) \quad y \leftarrow 3 \\
\} \text{ else \{ } \quad (2--) \quad \text{else \{ } \\
\quad y \leftarrow y - 2 \quad (2?1) \quad y \leftarrow y - 1 \\
\} \quad (2--) \} 
\]

\[
\text{post } = x\langle 1 \rangle + y\langle 1 \rangle = x\langle 2 \rangle + y\langle 2 \rangle - 1 
\]

then running

\[
\text{wp.} 
\]

produces the goal

Type variables: <none>

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

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

\[
x \ll M.g() \quad (1) \quad x \ll [1..10] 
\]

\[
\text{post } = \\
\quad \text{if } y\langle 2 \rangle = 0 \text{ then } \\
\quad \text{if } y\langle 1 \rangle = 0 \text{ then } x\langle 1 \rangle + 2 = x\langle 2 \rangle + 3 - 1 \\
\quad \text{else } x\langle 1 \rangle + (y\langle 1 \rangle - 2) = x\langle 2 \rangle + 3 - 1 \\
\quad \text{else } \\
\quad \quad \text{let } y_R = y\langle 2 \rangle - 1 \text{ in } \\
\quad \quad \text{if } y\langle 1 \rangle = 0 \text{ then } x\langle 1 \rangle + 2 = x\langle 2 \rangle + y_R - 1 \\
\quad \quad \text{else } x\langle 1 \rangle + (y\langle 1 \rangle - 2) = x\langle 2 \rangle + y_R - 1 
\]

Syntax: \text{wp } n_1 \ n_2. \text{ In pRHL, let wp consume } \text{exactly } n_1 \text{ statements of the first program and } n_2 \text{ statements of the second program. Fails if this isn’t possible.}

Syntax: \text{wp } n. \text{ In pHL and HL, let wp consume } \text{exactly } n \text{ statements of the program. Fails if this isn’t possible.}

\[ \circ \text{ rnd} \]

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} | \text{rnd } f | \text{rnd } f \ g. \text{ If the conclusion is a pRHL statement judgement whose programs end with random assignments } x_1 \ll d_1 \text{ and } x_2 \ll d_2, \text{ and } f \text{ and } g \text{ are functions between the types of } x_1 \text{ and } x_2, \text{ then consume those random assignments, replacing the conclusion’s postcondition by the probabilistic weakest precondition of the random assignments wrt. } f \text{ and } g. \text{ The new postcondition checks that:}

- \text{f and g are an isomorphism between the distributions } d_1 \text{ and } d_2;
• for all elements $u$ in the support of $d_1$, the result of substituting $u$ and $f(u)$ for $x_1(1)$ and $x_2(2)$ in the conclusion’s original postcondition holds.

When $g$ is $f$, it can be omitted. When $f$ is the identity, it can be omitted.

For example, if the current goal is

Type variables: <none>

n : int

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

pre = y(2) = n

\( x \in \{0,1\} \quad \Rightarrow \quad y \leftarrow y - 1 \)
\( x \in [2..3] \)

post = \( x(1) \leftrightarrow x(2) + y(2) = n + 2 \)

then running

\[
\text{rnd (fun b => b ? 3 : 2) (fun m => m = 3).}
\]

produces the goal

Type variables: <none>

n : int

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

pre = y(2) = n

\( y \leftarrow y - 1 \)

post =

\[
(\text{forall } (xR : \text{int}), \\
\quad xR \notin [2..3] \Rightarrow xR = \text{if } xR = 3 \text{ then } 3 \text{ else } 2) \&
\]

\[
(\text{forall } (xR : \text{int}), \\
\quad xR \notin [2..3] \Rightarrow \text{mu1 } [2..3] xR = \text{mu1 } \{0,1\} (xR = 3)) \&
\]

\[
\text{forall } (xL : \text{bool}), \\
\quad xL \notin \{0,1\} \Rightarrow \\
\quad (\text{if } xL \text{ then } 3 \text{ else } 2) \notin [2..3]) \&
\]

\[
\quad (xL \leftrightarrow (\text{if } xL \text{ then } 3 \text{ else } 2) + y(2) = n + 2)
\]

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

**Syntax:** \texttt{rnd\{1\} | rnd\{2\}}. If the conclusion is a PRHL statement judgement whose designated program (1 or 2) ends with a random assignment \(x \in d\), then consume that random assignment, replacing the conclusion’s postcondition with a check that:

• the weight of \(d\) is 1 (so the random assignment can’t fail);

• for all elements \(u\) in the support of \(d\), the result of substituting \(u\) for \(x(i)\)—where \(i\) is the selected side—in the conclusion’s original postcondition holds.

For example, if the current goal is
Type variables: <none>

&1 \text{(left)} : M.f [programs are in sync]
&2 \text{(right)} : M.f

pre = true

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

\text{post} = \{x\}

then running \( \text{rnd}(1) \).

produces the (false!) goal

Type variables: <none>

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

pre = true

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

\text{post} = \text{is\_lossless} \{0,1\} \& \text{forall} \ (x0 : \text{bool}) , \ x0 \ \text{in} \ \{0,1\} \Rightarrow x0 = x(2)

\textbf{Syntax:} \ \texttt{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) \( x \in [1..10] \)

\text{post} = 3 \leq x + y \leq 12

then running \( \text{rnd} \).

produces the goal

Type variables: <none>

Context : M.f

pre = true
(1) \( y \leftarrow 2 \)

\[
\text{post} = \forall (x_0 : \text{int}), x_0 \in [1..10] \Rightarrow 3 \leq x_0 + y \leq 12
\]

**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.

86

### 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>

\[
&1 (\text{left} ) : M.f \\
&2 (\text{right} ) : N.f \\
\text{pre} = \{x, y\} \\
\text{if} (y < x) \{  \\
 z \leftarrow x - y \\
\} \text{ else } \{  \\
 z \leftarrow y - x \\
\}  \\
z \leftarrow z + 2
\]

\[
\text{post} = 0 \leq z(1) \land z(1) = 2 * (z(2) - 1)
\]

then running

**if.**

produces the goals

Type variables: <none>

\[
\forall &1 &2, =\{x, y\} \Rightarrow y(1) < x(1) \iff y(2) < x(2)
\]

and

Type variables: <none>

\[
&1 (\text{left}) : M.f
\]
CHAPTER 3. TACTICS

&2 (right) : N.f

pre = ={x, y} \ / \ y{1} < x{1}

z := x - y

(1) z := x - y + 1

z := z * 2

(2)

post = 0 <= z{1} \ / \ z{1} = 2 * (z{2} - 1)

and

Type variables: <none>

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

pre = ={x, y} \ / \ ! y{1} < x{1}

z := y - x

(1) z := y - x + 1

z := z * 2

(2)

post = 0 <= z{1} \ / \ z{1} = 2 * (z{2} - 1)

Syntax: if{1} | if{2}. If the goal’s conclusion is a pRHL judgement in which the first statement of the specified program is an if statement, then reduce the goal to two subgoals:

• One where the if statement has been replaced by its “then” part, and the precondition has been augmented by the assertion that the if statement’s boolean expression is true in the specified program’s memory.

• One where the if statement has been replaced by its “else” part, and the precondition has been augmented by the assertion that the if statement’s boolean expression is false in the specified program’s memory.

For example, if the current goal is

Type variables: <none>

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

pre = ={x, y}

if (y < x) {

z := x - y

(1--) if (y < x) {

z := x - y + 1

} else {

z := y - x

(1?1) else {

z := y - x + 1

}

z := z * 2

(2--)}

post = 0 <= z{1} \ / \ z{1} = 2 * (z{2} - 1)

then running

if{1}.

produces the goals

Type variables: <none>
CHAPTER 3. TACTICS

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

pre = ={x, y} \& y(1) < x(1)

z \leftarrow x - y

\begin{align*}
(1--) & \textbf{if } (y < x) \{ \\
(1.1) & \quad z \leftarrow x - y + 1 \\
(1--) & \quad \textbf{else } \\
(1?1) & \quad z \leftarrow y - x + 1 \\
(1--) & \}
\end{align*}

z \leftarrow z \ast 2

(2--)

post = 0 \leq z(1) \& z(1) = 2 \ast (z(2) - 1)

and

Type variables: <none>

---

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

pre = ={x, y} \& y(1) < x(1)

z \leftarrow y - x

\begin{align*}
(1--) & \textbf{if } (y < x) \{ \\
(1.1) & \quad z \leftarrow x - y + 1 \\
(1--) & \quad \textbf{else } \\
(1?1) & \quad z \leftarrow y - x + 1 \\
(1--) & \}
\end{align*}

z \leftarrow z \ast 2

(2--)

post = 0 \leq z(1) \& z(1) = 2 \ast (z(2) - 1)

**Syntax:** \textbf{if}. If the goal’s conclusion is an HL judgement whose first statement is an \textbf{if} statement, then reduce the goal to two subgoals:

- One where the \textbf{if} statement has been replaced by its “then” part, and the precondition has been augmented by the assertion that the \textbf{if} statement’s boolean expression is true.
- One where the \textbf{if} statement has been replaced by its “else” part, and the precondition has been augmented by the assertion that the \textbf{if} statement’s boolean expression is false.

For example, if the current goal is

Type variables: <none>

---

Context : M.f

pre = true

\begin{align*}
(1--) & \textbf{if } (y < x) \{ \\
(1.1) & \quad z \leftarrow x - y \\
(1--) & \quad \textbf{else } \\
(1?1) & \quad z \leftarrow y - x \\
(1--) & \}
(2--) & \quad z \leftarrow z \ast 2
\end{align*}

post = 0 \leq z
then running

```c
if.
```

produces the goals

Type variables: <none>

---

Context : M.f
pre = y < x
(1) z <- x - y
(2) z <- z * 2
post = 0 <= z

and

Type variables: <none>

---

Context : M.f
pre = ! y < x
(1) z <- y - x
(2) z <- z * 2
post = 0 <= z

© while
Syntax: While I. Here I is an invariant (formula), which may reference variables of the two programs, interpreted in their memories. If the goal’s conclusion is a prhl statement judgement whose programs both end with while statements, reduce the goal to two subgoals whose conclusions are prhl statement judgements:

- One whose first and second programs are the bodies of the first and second while statements, whose precondition is the conjunction of I and the while statements’ boolean expressions (the first of which is interpreted in memory &1, and the second of which is interpreted in &2) and whose postcondition is the conjunction of I and the assertion that the while statements’ boolean expressions (interpreted in the appropriate memories) are equivalent.

- One whose precondition is the original goal’s precondition, whose first and second programs are all the results of removing the while statements from the two programs, and whose postcondition is the conjunction of:
  - the conjunction of I and the assertion that the while statements’ boolean expressions are equivalent; and
  - the assertion that, for all values of the variables modified by the while statements, if the while statements’ boolean expressions don’t hold, but I holds, then the original goal’s postcondition holds (in I, the while statements’ boolean expressions, and the postcondition, variables modified by the while statements are replaced by universally quantified identifiers; otherwise, the boolean expressions are interpreted in the program’s respective memories, and the memory references of I and the postcondition are maintained).

For example, if the current goal is
Type variables: <none>

n : int

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

pre = = (x, y) \ y(1) = n

\[
\begin{align*}
z &\leftarrow 0 & (1--) & z &\leftarrow 1 \\
\textbf{while} (0 < x) \{ & (2--) & i &\leftarrow 1 \\
z &\leftarrow z + 1 & (2.1) \\
x &\leftarrow x - 1 & (2.2) \\
\} & (2--) \\
(3--) & \textbf{while} (i <= x) \{ \\
(3.1) & z &\leftarrow z + 2 \\
(3.2) & i &\leftarrow i + 1 \\
(3--) & \}
\end{align*}
\]

post = (z(1) + y(1) - n) * 2 + 1 = z(2) + y(2) - n

then running

\[
\textbf{while} (x(1) - 1 = x(2) - i(2) /\ z(1) * 2 + 1 = z(2)).
\]

produces the goals

Type variables: <none>

n : int

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

pre =

\[
\begin{align*}
(x(1) - 1 = x(2) - i(2) /\ z(1) * 2 + 1 = z(2)) /\ \\
0 < x(1) /\ i(2) <= x(2)
\end{align*}
\]

\[
\begin{align*}
z &\leftarrow z + 1 & (1) & z &\leftarrow z + 2 \\
x &\leftarrow x - 1 & (2) & i &\leftarrow i + 1 \\
\end{align*}
\]

post =

\[
\begin{align*}
(x(1) - 1 = x(2) - i(2) /\ z(1) * 2 + 1 = z(2)) /\ \\
(0 < x(1) <= i(2) <= x(2))
\end{align*}
\]

and

Type variables: <none>

n : int

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

pre = = (x, y) \ y(1) = n

\[
\begin{align*}
z &\leftarrow 0 & (1--) & z &\leftarrow 1 \\
(2--) & i &\leftarrow 1 \\
\end{align*}
\]

post =

\[
\begin{align*}
((x(1) - 1 = x(2) - i(2) /\ z(1) * 2 + 1 = z(2)) /\ \\
0 < x(1) <= i(2) <= x(2))
\end{align*}
\]
\[ (0 < x_1 \Leftrightarrow i_2 <= x_2)) \land \]
\[
\text{forall} \ (x_L \ z_L \ i_R \ z_R : \text{int}),
\]
\[
! 0 < x_L \Rightarrow
! i_R <= x_2 \Rightarrow
x_L - 1 = x_2 - i_R \land z_L * 2 + 1 = z_R \Rightarrow
(z_L + y_1 - n) * 2 + 1 = z_R + y_2 - n
\]

**Syntax:** \texttt{while}({1}\text{ I v | while}({2}\text{ I v}). Here \textit{I} is an \textit{invariant} (formula) and \textit{v} is a \textit{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 \textit{pRHL} statement judgement whose designated program (1 or 2) ends with a \textit{while} statement, reduce the goal to two subgoals:

- One whose conclusion is a \textit{pHL} statement judgement, saying that running the body of the \textit{while} statement in a memory in which \textit{I} holds and the \textit{while} statement’s boolean expression is true is guaranteed to result in termination in a memory in which \textit{I} holds and in which the value of the variant expression \textit{v} has decreased by at least 1. (More precisely, the \textit{pHL} statement judgment is universally quantified by the memory of the non-designated program and the initial value of \textit{v}. References to the variables of the nondenominated program in \textit{I} and \textit{v} are interpreted in this memory; reference to the variables of the designed program have their memory references removed.)

- One whose conclusion is a \textit{pRHL} statement judgement whose precondition is the original goal’s precondition, whose designated program is the result of removing the \textit{while} statement from the original designated program, whose other program is unchanged, and whose postcondition is the conjunction of \textit{I} and the assertion that, for all values of the variables modified by the \textit{while} statement, that the conjunction of the following formulas holds:
  
  - the assertion that, if \textit{I} holds, but the variant expression \textit{v} is not positive, then the \textit{while} statement’s boolean expression is false;
  
  - the assertion that, if the \textit{while} statement’s boolean expression doesn’t hold, but \textit{I} holds, then the original goal’s postcondition holds.

For example, if the current goal is

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

```
&1 (left ) : N.f
&2 (right) : N.f
pre = ={n} \land 0 <= n{1}
x <- 0 \hspace{1cm} (1--)
i <- 0 \hspace{1cm} (2--)
while (i < n) {
  x <- x + (i + 1) \hspace{1cm} (3.1)
i <- i + 1 \hspace{1cm} (3.2)
}
post = x{1} <= n{2} * n{2}
```

then running

```
while({1} \ (0 <= i{1} <= n{1}) \land x{1} <= i(1) * i(1)) \ (n{1} - i{1}).
```

produces the goals

```
Type variables: <none>
```
forall \_ \ (z : int),
\begin{align*}
\text{phoarel} \ & 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] = 1%r
\end{align*}

and

Type variables: <none>

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

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

\begin{align*}
&\text{Type variables: <none>} \\
m : \text{int} \\
&\text{Context} : M. f \\
&\text{pre} = m = n \land 0 \leq n \\
&(1--) x \leftarrow 0 \\
&(2--) i \leftarrow 0 \\
&(3--) \text{while} \ (i < n) \{ \\
&(3.1) x \leftarrow x + (i + 1) \\
\end{align*}
(3.2) \[
i \leftarrow i + 1
\]
(3--) }

\text{post} = x \leq m \times m

then running
\textbf{while} (0 \leq i \leq n \land x \leq i \times i).

produces the goals

Type variables: <none>

\[m : \text{int}\]

Context : M.f

\text{pre} = \left(0 \leq i \leq n \land x \leq i \times i \land i < n\right)

(1) \[x \leftarrow x + (i + 1)\]
(2) \[i \leftarrow i + 1\]

\text{post} = 0 \leq i \leq n \land x \leq i \times i

and

Type variables: <none>

\[m : \text{int}\]

Context : M.f

\text{pre} = m = n \land 0 \leq n

(1) \[x \leftarrow 0\]
(2) \[i \leftarrow 0\]

\text{post} = \left(0 \leq i \leq n \land x \leq i \times i \land \forall (i_0 x_0 : \text{int}), ! i_0 < n \Rightarrow 0 \leq i_0 \leq n \land x_0 \leq i_0 \times i_0 \Rightarrow x_0 \leq m \times m\right.

\textbf{Syntax:} while I v. pHL version...

\textcopyright 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.

\textbf{Syntax:} call (_ : P \Rightarrow Q). If the goal’s conclusion is a pRHL statement judgement whose programs end with procedure calls or procedure call assignments (resp., an HL statement judgement whose program ends with a procedure call or procedure call assignment), then generate two subgoals:

- One whose conclusion is a pRHL judgement (resp., HL judgement) whose precondition is \(P\), whose procedures are the procedures being called (resp., procedure is the procedure being called), and whose postcondition is \(Q\).
- One whose conclusion is a pRHL statement judgement (resp., HL statement judgement) whose precondition is the original goal’s precondition, whose programs are (resp., program is) the result of removing the procedure calls (resp., call) from the programs (resp., program), and whose postcondition is the conjunction of
– the result of replacing the procedures’ (resp., procedure’s) parameter(s) by their actual argument(s) in \( P \); and
– the assertion that, for all values of the global variable(s) modified by the procedures (resp., procedure) and the results (resp., result) of the procedure calls (resp., procedure call), if \( Q \) holds (where these quantified identifiers have been substituted for the modified variables and procedure results), then the original goal’s postcondition holds (where the modified global variables and occurrences of the variables (resp., variable) (if any) to which the results of the procedure calls are (resp., result of the procedure call is) assigned have been replaced by the appropriate quantified identifiers).

For example, if the current goal is

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

and the procedures \( M.f \) and \( N.f \) have a single parameter, \( y \), then running

\[
\text{call} \ (\_ : \{y\} \land M.x\{1\} = -N.x\{2\} \implies \{\text{res}\} \land M.x\{1\} = -N.x\{2\})
\]

produces the goals

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

and

\[
\begin{align*}
& 1 \ (\text{left}) : M.g \\
& 2 \ (\text{right}) : N.g \\
& \text{pre} = \{u\} \\
M.x & \leftarrow 5 \quad (1) \ N.x \leftarrow -5 \\
\text{post} = \\
(\{u\} \land M.x\{1\} = -N.x\{2\}) \land \\
\forall \text{result}_L \text{ result}_R \ x_L \ x_R : \text{int}, \\
\text{result}_L = \text{result}_R \land x_L = -x_R \implies \\
\text{result}_L + x_L = \text{result}_R - x_R
\end{align*}
\]

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 \textit{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

\[
\text{lemma } M_N_f : \\
\text{equiv}[M.f - N.f : \\
\quad \forall y. M.x{1} = -N.x{2} \implies \{\text{res} \} \land M.x{1} = -N.x{2}].
\]

then running

\[
\text{call } M_N_f.
\]

produces the goal

Type variables: <none>

\[
\begin{align*}
& \&1 \text{ (left)} : M.g \\
& \&2 \text{ (right)} : N.g \\
& \text{pre} = \{u\} \\
& \text{M.x} \leftarrow 5 & (1) \text{ N.x} \leftarrow -5 \\
& \text{post} = \\
& \quad \forall \{u\} \land M.x{1} = -N.x{2}) \land \\
& \quad \forall \text{result_L result_R x_L x_R : int}, \\
& \quad \text{result_L} = \text{result_R} \land x_L = -x_R \\
& \quad \text{result_L} + x_L = \text{result_R} - x_R
\end{align*}
\]

Syntax: \text{call1} \{ \_ : P \implies Q \} \mid \text{call2} \{ \_ : P \implies Q \}. If the goal’s conclusion is a pRHL statement judgement whose designated program ends with a procedure call, then generate two subgoals:

- One whose conclusion is a pHL judgement whose precondition is $P$, whose procedure is the procedure being called, whose postcondition is $Q$, and whose bound part specifies equality with probability 1. (Consequently, $P$ and $Q$ may not mention $\&1$ and $\&2$.)

- One whose conclusion is a pRHL statement judgement whose precondition is the original goal's precondition, whose programs are the result of removing the procedure call from the designated program, and leaving the other program unchanged, and whose postcondition is the conjunction of
  
  - the result of replacing the procedure’s parameter(s) by their actual argument(s) in $P$; and
  
  - the assertion that, for all values of the global variable(s) modified by the procedure and the result of the procedure call, if $Q$ holds (where these quantified identifiers have been substituted for the modified variables and procedure result), then the original goal’s postcondition holds (where the modified global variables and occurrences the variable (if any) to which the result of the procedure call is assigned have been replaced by the appropriate quantified identifiers).

For example, if the current goal is

Type variables: <none>

\[
\begin{align*}
x2 : \text{int} \\
\&1 \text{ (left)} : M.g \\
\&2 \text{ (right)} : N.g \\
\text{pre} = x2 = N.x{2} \land N.x{2} = u{2} \land u{1} \equiv 2 = u{2} \equiv 2
\end{align*}
\]
CHAPTER 3. TACTICS

\[ M_x \leftarrow u \]  \hspace{1cm} (1)
\[ M_f(7) \]  \hspace{1cm} (2)

\[
\text{post} = (M \cdot x \% 2 = 0) = (N \cdot x \% 2 = 0)
\]

then running

\texttt{call(1)} \_ : M \cdot x \% 2 = x2 \% 2 \implies M \cdot x \% 2 = x2 \% 2).

produces the goals

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

\[ x2 : \text{int} \]

\[
\text{pre} = M \cdot x \% 2 = x2 \% 2
\]

\[ M \cdot f \]

\[ [\,] \implies 1 \%
\]

\[
\text{post} = M \cdot x \% 2 = x2 \% 2
\]

and

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

\[ x2 : \text{int} \]

\[ \&1 \text{ (left) : M \cdot g} \]

\[ \&2 \text{ (right) : N \cdot g} \]

\[
\text{pre} = x2 = N \cdot x(2) \land N \cdot x(2) = u(2) \land u(1) \% 2 = u(2) \% 2
\]

\[ M \cdot x \leftarrow u \]  \hspace{1cm} (1)

\[
\text{post} = M \cdot x(1) \% 2 = x2 \% 2 & x
\]

\[ \text{forall} \ (x_L : \text{int}), \]

\[ x_L \% 2 = x2 \% 2 \implies (x_L \% 2 = 0) = (N \cdot x(2) \% 2 = 0) \]

Alternatively, a proof term whose conclusion is a \texttt{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

\texttt{lemma M\_f (z : \text{int}) : p\text{hoare} [M \cdot f : M \cdot x \% 2 = z \% 2 \implies M \cdot x \% 2 = z \% 2] = 1\%r}.

then running

\texttt{call(1)} \ (M \cdot f \ x2).

produces the goal

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

\[ x2 : \text{int} \]

\[ \&1 \text{ (left) : M \cdot g} \]

\[ \&2 \text{ (right) : N \cdot g} \]
pre = x2 = N.x(2) \ N.x(2) = u(2) \ u(1) \ u(2) = u(2)

M.x <- u  \ N.x <- u(1) \ N.x = u(2)

post =
M.x{1} \ N.x{2} = x2 \ M.x{1} \ N.x{2} = x2 => (x_L \ N.x{2} = 0) = (N.x{2} = 0)

Syntax: \ call \ (_ : I) \ If the conclusion is a pRHL statement judgement whose programs end with calls to concrete procedures (resp., an HL statement judgement whose program ends with a call to a concrete procedure), then use the specification argument to call generated from the invariant I, and automatically apply proc to its first subgoal. In the pRHL case, its precondition will assume equality of the procedures’ parameters, and its postcondition will assert equality of the results of the procedure calls.

For example, if the current goal is

Type variables: <none>

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

pre = =(u)
M.x <- 5  \ N.x <- -5
z <= M.f(u)  \ z <= N.f(u)

post = z(1) = M.x(1) = z(2) = N.x(2)

and modules M and N contain

\ var \ x : int
\ proc f(y : int) : int = {
\ x <- x + y;
\ return x;
\ }

and

\ var \ x : int
\ proc f(y : int) : int = {
\ x <- x - y;
\ return -x;
\ }

respectively, then running

call (_ : M.x(1) = -N.x(2)).

produces the goals

Type variables: <none>

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

pre = =y  \ M.x(1) = -N.x(2)
M.x <- M.x + y  \ N.x <- N.x - y
post = M.x{1} = -N.x{2} \land M.x{1} = -N.x{2}

and

Type variables: <none>

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

pre = ={u}
M.x <- 5
N.x <- -5

post =
(={u} \land M.x{1} = -N.x{2}) \land
\forall (result_L result_R x_L x_R : int),
result_L = result_R \land x_L = -x_R \Rightarrow
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) : N(Adv).h
&2 (right) : N(Adv).h

pre = =\{y, glob Adv\} \land Or.x{1} \mod 2 = 0 \land Or.x{2} \mod 2 = 0

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

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

then running

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

produces the goals

Type variables: <none>

Adv: T{Or}

pre = true \land Or.x{1} \mod 2 = 0 \land Or.x{2} \mod 2 = 0

Or.f1 = Or.f1

post = =\{res\} \land Or.x{1} \mod 2 = 0 \land Or.x{2} \mod 2 = 0

and

Type variables: <none>

Adv: T{Or}

pre = true \land Or.x{1} \mod 2 = 0 \land Or.x{2} \mod 2 = 0

Or.f2 = Or.f2

post = =\{res\} \land Or.x{1} \mod 2 = 0 \land Or.x{2} \mod 2 = 0

Syntax: call (_ : B, I). If the conclusion is a prHL statement judgement whose programs end with calls of the same abstract procedure, then use the specification argument to call generated from the bad event B and invariant I, and automatically apply proc B I to its first subgoal, pruning the first two subgoals the application generates, because their conclusions consist of ambient logic formulas that are true by construction, and pruning the next goal (showing the losslessness of the abstract procedure given the losslessness of the abstract oracles it uses), if trivial suffices to solve it. The specification’s precondition will assume equality of the procedure’s parameters and of the global variables of the module containing the procedure as well as I, and its postcondition will assert I and the equality of the results of the procedure calls and of the global variables of the containing module—but only when B does not hold.

For example, given the declarations

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

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

module Or1 : OR = {
  var qry, rsp : int
CHAPTER 3. TACTICS

var queried : bool

proc init() = {
    qry <$ [low .. upp]; rsp <$ [low .. upp];
    queried <- false;
}

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

}. module Or2 : OR = {
    var qry : int
    var queried : bool
    proc init() = {
        qry <$ [low .. upp];
        queried <- false;
    }
    proc qry(x : int) : int = {
        var y : int;
        y <$ [low .. upp];
        queried <- queried \ x = qry;
        return y;
    }
}. module type ADV(O : OR) = {
    proc * f() : bool {O.qry}
}. if the current goal is

Type variables: <none>

Adv: ADV{Or1, Or2}
11_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)

b <$> Adv(Or1).f() (1) b <$> Adv(Or2).f()

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

then running

call (_, Or2.queried, (Or1.qry(1) = Or2.qry(2))).

produces the goals
CHAPTER 3. TACTICS

Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
  islossless 0.qry => islossless Adv(0).f

pre = !Or2.queried(2) /\ ={x} /\ Or1.qry(1) = Or2.qry(2)

Or1.qry - Or2.qry

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

and

Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
  islossless 0.qry => islossless Adv(0).f

forall &2, Or2.queried(2) => islossless Or1.qry

and

Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
  islossless 0.qry => islossless Adv(0).f

forall _,
  phoare[ Or2.qry : Or2.queried /\ true ==> Or2.queried /\ true] = 1%

and

Type variables: <none>

Adv: ADV{Or1, Or2}
ll_Adv_f: forall (O <: OR{Adv}),
  islossless 0.qry => islossless Adv(0).f

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

pre = Or1.qry(1) = Or2.qry(2)

post =

(!Or2.queried(2) => true /\ Or1.qry(1) = Or2.qry(2)) &&
forall (result_L result_R : bool) (Adv_L Adv_R : (glob Adv))
  (queried_R : bool),
  (!queried_R => result_L = result_R /\ Adv_L = Adv_R /\ Or1.qry(1) = Or2.qry(2)) =>
  !queried_R => result_L = result_R

Syntax: call (_ : B, I, J). If the conclusion is a pRHL statement judgement whose programs end with calls of the same abstract procedure, then use the specification argument to call generated from the bad event B and invariants I and J, and automatically apply proc B I J to its first subgoal, pruning the first two subgoals the application generates, because their conclusions
consist of ambient logic formulas that are true by construction, and pruning the next goal (showing the losslessness of the abstract procedure given the losslessness of the abstract oracles it uses), if trivial suffices to solve it. The specification’s precondition will assume equality of the procedure’s parameters and of the global variables of the module containing the procedure as well as \( I \), and its postcondition will assert

- \( I \) and the equality of the results of the procedure calls and of the global variables of the containing module—if \( B \) does not hold; and
- \( J \)—if \( B \) does hold.

For example, given the declarations of the preceding example if the current goal is

Type variables: <none>

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

ll_Adv_f: \[ \forall (O : OR\{Adv\}), \]

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

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

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

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

\[ b <@ \text{Adv(Or1).f()} \quad (1) \quad b <@ \text{Adv(Or2).f()} \]

\[ \text{post = Or1.queried(1) = Or2.queried(2) /\ (!Or2.queried(2) \Rightarrow ={b})} \]

then running

\[ \text{call (}: \]

\[ \text{Or2.queried,} \]

\[ (\text{Or1.qry(1) = Or2.qry(2) /\ Or1.queried(1) = Or2.queried(2)}, \]

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

produces the goals

Type variables: <none>

Adv: ADV\{Or1, Or2\}

ll_Adv_f: \[ \forall (O : OR\{Adv\}), \]

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

\[ \text{pre = !Or2.queried(2) /\} \]

\[ ={x} /\ Or1.qry(1) = Or2.qry(2) /\ Or1.queried(1) = Or2.queried(2) \]

\[ \text{Or1.qry - Or2.qry} \]

\[ \text{post = if Or2.queried(2) then Or1.queried(1) = Or2.queried(2) \}

\[ \text{else} \]

\[ ={\text{res}} /\ Or1.qry(1) = Or2.qry(2) /\ Or1.queried(1) = Or2.queried(2) \]

and

Type variables: <none>

Adv: ADV\{Or1, Or2\}
3.4.2 Tactics for Transforming Programs

Unless otherwise specified, the tactics of this subsection only apply to goals whose conclusions are PRHL, PHL or HL statement judgements, reducing such a goal to a single subgoal in which only the program(s) of those statement judgements have changed.

Many of these tactics take code positions consisting of a sequence of positive numerals separated by dots. E.g., 2.1.3 says to go to the statement 2 of the program, then to substatement 1 of it, then to sub-substatement 3 of it. We use the variable c to range over code positions.
\[\text{swap}\]

All versions of the tactic work for PRHL (an optional side can be given), PHL and HL statement judgements. We’ll describe their operation in terms of a single program (list of statements).

**Syntax:** \(\text{swap } n \ b \ l\). Fails unless \(1 \leq n < m \leq l\) and the program has at least \(l\) statements. Swaps the statement block from positions \(n\) through \(m - 1\) with the statement block from \(m\) through \(l\), failing if these blocks of statements aren’t independent.

**Syntax:** \(\text{swap } [n..m] \ k\). Fails unless \(1 \leq n \leq m\) and the program has at least \(m\) statements.

- If \(k\) is non-negative, move the statement block from \(n\) through \(m\) forward \(k\) positions, failing if the program doesn’t have at least \(m + k\) statements or if the swapped statements blocks aren’t independent.

- If \(k\) is negative, move the statement block from \(n\) through \(m\) backward \(−k\) positions, failing if \(n + k < 1\) or if the swapped statement blocks aren’t independent.

**Syntax:** \(\text{swap } n \ k\). Equivalent to \(\text{swap } [n..n] \ k\).

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

For example, suppose the current goal is

Type variables: <none>

\(k1\) (left) : M.f [programs are in sync]
\(k2\) (right) : M.f

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

(1) M.x \leftarrow \text{true}
(2) M.y \leftarrow \text{false}
(3) M.z \leftarrow \text{true}
(4) M.w \leftarrow \text{false}

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

Then running

\[\text{swap } 1 \ 3 \ 4.\]

produces goal

Type variables: <none>

\(k1\) (left) : M.f [programs are in sync]
\(k2\) (right) : M.f

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

(1) M.z \leftarrow \text{true}
(2) M.w \leftarrow \text{false}
(3) M.x \leftarrow \text{true}
(4) M.y \leftarrow \text{false}

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

From which running

\[\text{swap } 2 \ 2.\]
produces goal

Type variables: <none>

&1 (left) : M.f [programs are in sync]
&2 (right) : M.f

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

(1) M.z ← true
(2) M.x ← true
(3) M.y ← false
(4) M.w ← false

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

From which running

swap(1) [3 .. 4] -1.

produces goal

Type variables: <none>

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

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

M.z ← true (1) M.z ← true
M.y ← false (2) M.x ← true
M.w ← false (3) M.y ← false
M.x ← true (4) M.w ← false

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

From which running

swap 2.

produces goal

Type variables: <none>

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

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

M.y ← false (1) M.x ← true
M.w ← false (2) M.y ← false
M.z ← true (3) M.z ← true
M.x ← true (4) M.w ← false

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

From which running

swap(2) -1.
produces goal

Type variables: <none>

\&1 (left) : M.f
\&2 (right) : M.f
pre = ={M.x, M.y, M.z, M.w}
M.y <- false
(1) M.x <- true
M.w <- false
(2) M.y <- false
M.z <- true
(3) M.w <- false
M.x <- true
(4) M.z <- true
post = ={M.x, M.y, M.z, M.w}

\section*{inline}

\textbf{Syntax:} \texttt{inline }$M_1.p_1 \ldots M_n.p_n$. Inline the selected \textit{concrete} procedures in both programs, with \texttt{pRHL}, and in the program, with \texttt{HL} and \texttt{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.

\textbf{Syntax:} \texttt{inline(1) }$M_1.p_1 \ldots M_n.p_n$ \texttt{| inline(2) }$M_1.p_1 \ldots M_n.p_n$. Do the inlining in just the first or second program, in the \texttt{pRHL} case.

\textbf{Syntax:} \texttt{inline* | inline(1)* | inline(2)*}. Inline all concrete procedures, continuing until no more inlining is possible.

\textbf{Syntax:} \texttt{inline \texttt{occs} }$M.p$ \texttt{| inline(1) \texttt{occs} }$M.p$ \texttt{| inline(2) \texttt{occs} }$M.p$. Inline just the specified occurrences of $M.p$, where \texttt{occs} is a parenthesized nonempty sequence of positive numbers ($n_1 \ldots n_l$). E.g., (1 3) means the first and third occurrences of the procedure. In the \texttt{pRHL} case, a side \{1\} or \{2\} must be specified.

For example, given the declarations

\begin{verbatim}
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;
  }
}.
\end{verbatim}

if the current goal is

Type variables: <none>
\&1 \textbf{(left)} : M.h [programs are in sync]
\&2 \textbf{(right)} : M.h

pre = \{x\}

(1) \ b <\emptyset \ M.g(x + 1)

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

then running

\textbf{inline} M.g.

produces the goal

Type variables: <none>

\&1 \textbf{(left)} : M.h [programs are in sync]
\&2 \textbf{(right)} : M.h

pre = \{x\}

(1) \ x_0 \leftarrow x + 1
(2) \ M.y <\emptyset \ M.f(x_0 - 1)
(3) \ b \leftarrow x_0 + M.y + 1 = 3

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

From which running

\textbf{inline} (2) M.f.

produces the goal

Type variables: <none>

\&1 \textbf{(left)} : M.h
\&2 \textbf{(right)} : M.h

pre = \{x\}

x_0 \leftarrow x + 1 \quad \begin{array}{l}
(1) \ x_0 \leftarrow x + 1

M.y <\emptyset \ M.f(x_0 - 1) \quad (2) \ x_1 \leftarrow x_0 - 1

b \leftarrow x_0 + M.y + 1 = 3 \quad (3) \ x_1 \leftarrow x_1 + 1

(4) \ M.y \leftarrow x_1 \cdot 2

(5) \ b \leftarrow x_0 + M.y + 1 = 3

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

From which running

\textbf{inline} M.f.

produces the goal

Type variables: <none>

\&1 \textbf{(left)} : M.h [programs are in sync]
§2 (right) : M.h

pre = =(x)

(1) x0 ← x + 1
(2) x1 ← x0 - 1
(3) x1 ← x1 + 1
(4) M.y ← x1 + 2
(5) b ← x0 + M.y + 1 = 3

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

And, if the current goal is

Type variables: <none>

§1 (left) : M.h [programs are in sync]
§2 (right) : M.h

pre = =(x)

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

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

then running

*inline*. produces the goal

Type variables: <none>

§1 (left) : M.h [programs are in sync]
§2 (right) : M.h

pre = =(x)

(1) x0 ← x + 1
(2) x1 ← x0 - 1
(3) x1 ← x1 + 1
(4) M.y ← x1 + 2
(5) b ← x0 + M.y + 1 = 3

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

© rcondt

Syntax: rcondt n. If the goal’s conclusion is an HL statement judgement whose 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 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
CHAPTER 3. TACTICS

Type variables: <none>

Context : M.f

pre = true

(1--) x <\{0,1\}
(2--) y ← x "" x
(3--) if (!y) {
    (3.1) z ← true
    (3--) }
else {
    (3?1) z ← false
    (3--) }

post = z

then running

rcondt 3.

produces the goals

Type variables: <none>

Context : M.f

pre = true

(1) x <\{0,1\}
(2) y ← x "" x

post = !y

and

Type variables: <none>

Context : M.f

pre = true

(1) x <\{0,1\}
(2) y ← x "" x
(3) z ← true

post = z

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

- One whose conclusion is an HL statement judgement whose precondition is the original goal’s precondition, program is the first \( n-1 \) statements of the original goal’s designated program, and postcondition is the boolean expression of the if statement. Actually, the HL statement judgement is universally quantified by a memory of the non-designated program, and references in the precondition to variables of the non-designated program are interpreted in that memory.
• One whose conclusion is a pRHL statement judgement that’s the same as that of the original goal except that the if statement has been replaced by its then part.

For example, if the current goal is

Type variables: <none>

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

pre = N.x(1)
(1--) x <$> {0,1}  
(2--) y <-> x ^^ x  
(3--) if (!y) {  
(3.1) z <-> true  
(3--) } else {  
(3?) z <-> false  
(3--) }

post = N.x(1) = z(2)

then running

rcondt{2} 3.

produces the goals

Type variables: <none>

\forall \&m, hoare[ x <$> {0,1}; y <-> x ^^ x : N.x(m) => !y]

and

Type variables: <none>

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

pre = N.x(1)
(1) x <$> {0,1}  
(2) y <-> x ^^ x  
(3) z <-> true

post = N.x(1) = z(2)

Syntax: rcondt n. If the goal’s conclusion is an HL statement judgement whose nth statement is a while statement, reduce the goal to two subgoals.

• One whose 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 while statement.

• One whose conclusion is an HL statement judgement that’s the same as that of the original goal except that the while statement has been replaced by its body.

For example, if the current goal is
Type variables: <none>

Context : M.f

pre = true

(1--) i <- 0
(2--) while (i < 10) {
(2.1) i <- i + 1
(2--) }
(3--) while (i < 20) {
(3.1) i <- i + 2
(3--) }

post = i = 20

then running

\texttt{rcondt 3.}

produces the goals

Type variables: <none>

Context : M.f

pre = true

(1--) i <- 0
(2--) while (i < 10) {
(2.1) i <- i + 1
(2--) }

post = i < 20

and

Type variables: <none>

Context : M.f

pre = true

(1--) i <- 0
(2--) while (i < 10) {
(2.1) i <- i + 1
(2--) }
(3--) i <- i + 2
(4--) while (i < 20) {
(4.1) i <- i + 2
(4--) }

post = i = 20

Syntax: \texttt{rcondt\{1\} n} \mid \texttt{rcondt\{2\} n}. If the goal’s conclusion is a \texttt{pRHL} statement judgement where the nth statement of the designated program is a \texttt{while} statement, reduce the goal to two subgoals.
• One whose conclusion is an HL statement judgement whose precondition is the original goal’s precondition, program is the first \(n - 1\) statements of the original goal’s designated program, and postcondition is the boolean expression of the while statement. Actually, the HL statement judgement is universally quantified by a memory of the non-designated program, and references in the precondition to variables of the non-designated program are interpreted in that memory.

• One whose conclusion is a pRHL statement judgement that’s the same as that of the original goal except that the while statement has been replaced by its body.

For example, if the current goal is

\[
\begin{align*}
\&1 \text{(left) } & : M.f \\
\&2 \text{(right) } & : N.f \\
pre & = \text{true} \\
i & < - 0 \quad \text{(1--)} & i & < - 0 \\
\text{while} \ (i < 10) \ {\{ \} } & \quad \text{(2--)} & \text{while} \ (i < 10) \ {\{ \} } \\
i & < - i + 1 \quad \text{(2.1)} & i & < - i + 1 \\
\} & \quad \text{(2--)} & \} \\
\text{while} \ (i < 20) \ {\{ \} } & \quad \text{(3--)} & i & < - i + 2 \\
i & < - i + 2 \quad \text{(3.1)} & (3--)} \\
\} & \quad \text{(3--)} & \} \\
\} & \quad \text{(4--)} & \} \\
\text{while} \ (i < 20) \ {\{ \} } & \quad \text{(4.1)} & i & < - i + 2 \\
i & < - i + 2 \quad \text{(4.1)} & (4--)} \\
\} & \quad \text{(4--)} & \} \\
\} & \quad \text{(4--)} & \}
\end{align*}
\]

\[
\begin{align*}
\text{post} & = \{ i \}
\end{align*}
\]

then running

\[
rcondt{1} 3.
\]

produces the goals

\[
\begin{align*}
\text{forall } _, \ \text{hoare}[ i < - 0; \ \text{while} \ (i < 10) \ {\{ \} } : \text{true} \implies i < 20]
\end{align*}
\]

and

\[
\begin{align*}
\text{forall } _, \ \text{hoare}[ i < - 0; \ \text{while} \ (i < 10) \ {\{ \} } : \text{true} \implies i < 20]
\end{align*}
\]
CHAPTER 3. TACTICS

post = \{i\}

\[ \text{Syntax: } \texttt{rcondf } n. \text{ If the goal’s conclusion is an HL statement judgement whose n}^{th} \text{ statement is an } \texttt{if} \text{ statement, reduce the goal to two subgoals.} \]

- One whose conclusion is an HL statement judgement whose precondition is the original goal’s precondition, program is the first } n - 1 \text{ statements of the original goal’s program, and postcondition is the negation of the boolean expression of the } \texttt{if} \text{ statement.}

- One whose conclusion is an HL statement judgement that’s the same as that of the original goal except that the } \texttt{if} \text{ statement has been replaced by its } \texttt{else} \text{ part.

For example, if the current goal is

\begin{verbatim}
Type variables: <none>

Context : M.f
pre = true
(1--) x <$ \{0,1\}
(2--) y <- x ^^ x
(3--) \texttt{if} (y) {
(3.1) z <- true
(3--) }
(3?1) z <- false
(3--) }
post = !z
\end{verbatim}

then running

\[ \texttt{rcondf 3.} \]

produces the goals

\begin{verbatim}
Type variables: <none>

Context : M.f
pre = true
(1) x <$ \{0,1\}
(2) y <- x ^^ x
post = !y
\end{verbatim}

and

\begin{verbatim}
Type variables: <none>

Context : M.f
pre = true
\end{verbatim}
CHAPTER 3. TACTICS

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

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

For example, if the current goal is

\begin{verbatim}
Type variables: <none>

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

pre = !N.x{1}

(1--) x <\{0,1\}
(2--) y <- x ^^ x
(3--) if (y) {
    (3.1) z <- true
    (3--) }
else {
    (3?1) z <- false
    (3--) }

post = N.x{1} = z{2}
\end{verbatim}

then running

\begin{verbatim}
\texttt{rcondf}\{2\} 3.
\end{verbatim}

produces the goals

\begin{verbatim}
Type variables: <none>

\forall \&m, \mathit{hoare}[ x <\{0,1\}; y <- x ^^ x : !N.x{m} ==> !y]
\end{verbatim}

and

\begin{verbatim}
Type variables: <none>

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

pre = !N.x{1}
\end{verbatim}
Syntax: $\text{rcondf } n$. If the goal’s conclusion is an HL statement judgement whose $n$th statement is a while statement, reduce the goal to two subgoals.

- One whose conclusion is an HL statement judgement whose precondition is the original goal’s precondition, program is the first $n - 1$ statements of the original goal’s program, and postcondition is the negation of the boolean expression of the while statement.

- One whose conclusion is an HL statement judgement that’s the same as that of the original goal except that the while statement has been removed.

For example, if the current goal is

$$\begin{align*}
(1) & \quad x < \{0,1\} \\
(2) & \quad y \gets x \wedge x \\
(3) & \quad z \gets \text{false}
\end{align*}$$

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

then running

$$\text{rcondf } 3.$$ produces the goals

$$\begin{align*}
\text{Type variables: } & \langle \text{none} \rangle \\
\text{Context : } & M.f \\
\text{pre} = & \text{true} \\
(1--) & \quad i < 0 \\
(2--) & \quad \text{while } (i < 10) \{ \\
(2.1) & \quad i \gets i + 1 \\
(2--) & \} \\
(3--) & \quad \text{while } (i < 10) \{ \\
(3.1) & \quad i \gets i + 2 \\
(3--) & \}
\end{align*}$$

$$\text{post} = i = 10$$

and

$$\begin{align*}
\text{Type variables: } & \langle \text{none} \rangle \\
\text{Context : } & M.f \\
\text{pre} = & \text{true} \\
(1--) & \quad i < 0 \\
(2--) & \quad \text{while } (i < 10) \{ \\
(2.1) & \quad i \gets i + 1 \\
(2--) & \}
\end{align*}$$

$$\text{post} = \neg i < 10$$

$$\begin{align*}
\text{Type variables: } & \langle \text{none} \rangle
\end{align*}$$
CHAPTER 3. TACTICS

Context: M.f

pre = true

(1--) i <- 0
(2--) while (i < 10) {
(2.1) i <- i + 1
(2--) }

post = i = 10

Syntax: rcondf \{i\} n | rcondf \{2\} n. If the goal’s conclusion is a pRHL statement judgement where the n-th statement of the designated program is a while statement, reduce the goal to two subgoals.

- One whose conclusion is an HL statement judgement whose precondition is the original goal’s precondition, program is the first \( n - 1 \) statements of the original goal’s designated program, and postcondition is the negation of the boolean expression of the while statement. Actually, the HL statement judgement is universally quantified by a memory of the non-designated program, and references in the precondition to variables of the non-designated program are interpreted in that memory.
- One whose conclusion is a pRHL statement judgement that’s the same as that of the original goal except that the while statement has been removed.

For example, if the current goal is

Type variables: <none>

\( \&1 \text{ (left ) : M.f} \)
\( \&2 \text{ (right ) : N.f} \)
pre = true

\( i \gets 0 \)
(1--) \( i \gets 0 \)
(2--) \( \text{while} \ (i < 10) \{ \)
(2.1) \( i \gets i + 1 \)
(2--) \}
(3--) \( \text{while} \ (i < 10) \{ \)
(3.1) \( i \gets i + 2 \)
(3--) \}

post = \{i\}

then running

\( \text{rcondf} \{i\} 3. \)

produces the goals

Type variables: <none>

\( \text{forall} \_ , \text{hoare}[ i \gets 0; \text{while} \ (i < 10) \{ \_ \} : \text{true} \implies \neg i < 10] \)

and

Type variables: <none>
\begin{itemize}
\item $\&1$ (left): $M.f$
\item $\&2$ (right): $N.f$
\end{itemize}

$pre = true$

\begin{align*}
&i <- 0 \\
&\textbf{while} (i < 10) \{ \\
&\ \ i <- i + 1 \\
&\} \\
\end{align*}

post = $\{i\}$

---

\textbf{unroll}

\textbf{Syntax: unroll} $c$. If the goal's conclusion is an HL statement judgement whose $c$th statement is a \texttt{while} statement, then insert before that statement an \texttt{if} statement whose boolean expression is the \texttt{while} statement's boolean expression, whose \texttt{then} part is the \texttt{while} statements's body, and whose \texttt{else} part is empty.

For example, if the current goal is

\begin{itemize}
\item Type variables: <none>
\item Context : $M.f$
\item $pre = true$
\item (1--) $x <- 0$
\item (2--) $z <- 0$
\item (3--) \texttt{while} $(x < y)$ \{
\item (3.1) $z <- z + x$
\item (3.2) $x <- x + 1$
\item (3--) \}
\item post = $0 <= z$
\end{itemize}

then running \texttt{unroll 3.} produces the goal

\begin{itemize}
\item Type variables: <none>
\item Context : $M.f$
\item $pre = true$
\item (1--) $x <- 0$
\item (2--) $z <- 0$
\item (3--) \texttt{if} $(x < y)$ \{
\item (3.1) $z <- z + x$
\item (3.2) $x <- x + 1$
\item (3--) \}
\item (4--) \texttt{while} $(x < y)$ \{
\item (4.1) $z <- z + x$
\item (4.2) $x <- x + 1$
\item (4--) \}
\item post = $0 <= z$
\end{itemize}
And, if the current goal is

Type variables: <none>

Context : M.f

pre = true

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

post = 0 <= y

then running

unroll 1.2.

produces the goal

Type variables: <none>

Context : M.f

pre = true

(1----) if (0 <= y) {
(1.1--) x <- 0
(1.2--) if (x <> y) {
(1.2.1) x <- x + 1
(1.2--) }
(1.3--) while (x <> y) {
(1.3.1) z <- z + x
(1.3--) }
(1.4--) }

post = 0 <= y

Syntax: \texttt{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 \texttt{while} statement, then insert before that statement an \texttt{if} statement whose boolean expression is the \texttt{while} statement’s boolean expression, whose then part is the \texttt{while} statements’s body, and whose else part is empty.

For example, if the current goal is

Type variables: <none>

&1 (left ) : M.f [programs are in sync]
&2 (right ) : M.f

pre = ={y}

(1--) x <- 0
(2--) z <- 0
(3--) while (x < y) {
(3.1) z <- z + x
CHAPTER 3. TACTICS

(3.2) \( x \leftarrow x + 1 \)
(3--) }

post = \{z\}

then running

unroll{1} 3.

produces the goal

Type variables: <none>

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

pre = \{y\}

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

post = \{z\}

from which running

unroll{2} 3.

produces the goal

Type variables: <none>

\&1 (left) : M.f \quad \text{[programs are in sync]} \\
\&2 (right) : M.f

pre = \{y\}

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

post = \{z\}

splitwhile
CHAPTER 3. TACTICS

Syntax: `splitwhile c : e`. If the goal’s conclusion is an HL statement judgement whose \( c \)th statement is a `while` statement and \( e \) is a well-typed boolean expression in the `while` statement’s context, then insert before the `while` statement a copy of the `while` statement in which \( e \) is added as a conjunct of the statement’s boolean expression.

For example, if the current goal is

```
Type variables: <none>

Context : M.f

pre = true

(1--) x <- 0
(2--) z <- 0
(3--) while (x < y) {
  (3.1) z <- z + x
  (3.2) x <- x + 1
  (3--) }

post = 0 <= z
```

then running

```
splitwhile 3 : z <= 20.
```

produces the goal

```
Type variables: <none>

Context : M.f

pre = true

(1--) x <- 0
(2--) z <- 0
(3--) while (x < y /\ z <= 20) {
  (3.1) z <- z + x
  (3.2) x <- x + 1
  (3--) }

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

post = 0 <= z
```

Syntax: `splitwhile{1} c : e | splitwhile{2} c : e`. If the goal’s conclusion is a pRHL statement judgement where the \( c \)th statement of the designated program is a `while` statement and \( e \) is a well-typed boolean expression in the `while` statement’s context, then insert before the `while` statement a copy of the `while` statement in which \( e \) is added as a conjunct of the statement’s boolean expression.

For example, if the current goal is

```
Type variables: <none>

&1 (left) : M.f [programs are in sync]
&2 (right) : M.f
```
CHAPTER 3. TACTICS 121

pre = ={y}

(1--) x ← 0
(2--) z ← 0
(3--) while (x < y) {
(3.1) z ← z + x
(3.2) x ← x + 1
(3--}

post = ={z}

then running

\texttt{splitwhile(2) 3 : z <= 20.}

produces the goal

Type variables: <none>

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

pre = ={y}

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

post = ={z}

from which running

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

produces the goal

Type variables: <none>

\[\&1 \left( \text{left} \right) : M.f \text{ [programs are in sync]} \]
\[\&2 \left( \text{right} \right) : M.f \]

pre = ={y}

(1--) x ← 0
(2--) z ← 0
(3--) while (x < y / z <= 20) {
(3.1) z ← z + x
(3.2) x ← x + 1
(3--) }
(4--) while (x < y) {
(4.1) z ← z + x
(4.2) x ← x + 1
(4--) }

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

CHAPTER 3. TACTICS

(4--) }
post = {z}

§ fission

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

Let
• s_1 be the l statements before the while statement at position c;
• e be the boolean expression of the while statement;
• s_2 be the first m statements of the body of the while statement;
• s_3 be the next n - m statements of the body of the while statement;
• s_4 be the rest of the body of the while statement.

Fails unless:
• e doesn’t reference the variables written by s_2 and s_3;
• s_1 and s_4 don’t read or write the variables written by s_2 and s_3;
• s_2 and s_3 don’t write the variables written by s_1 and s_4;
• s_2 and s_3 don’t read or write the variables written by the other.

The tactic replaces
s_1 while (e) \{ s_2 s_3 s_4 \}
by
s_1 while (e) \{ s_2 s_4 \}
s_1 while (e) \{ s_3 s_4 \}

For example, if the current goal is

Type variables: <none>

Context : M.f

pre = 0 < n

(1-- ) x <- 0
(2-- ) y <- 0
(3-- ) i <- 0
(4-- ) j <- 0
(5-- ) while (i + j < n) {
(5.1) x <- x * i
(5.2) x <- x + (j + 1)
(5.3) y <- y * j
(5.4) y <- y + (i + 2)
(5.5) i <- i + 1
(5.6) j <- j + 2
(5--) }
post = 0 < x + y
then running

\[
\text{fission} \ 5!2 \ @ \ 2, \ 4.
\]

produces the goal

Type variables: <none>

Context : M.f

pre = 0 < n

(1--) \( x \leftarrow 0 \)
(2--) \( y \leftarrow 0 \)
(3--) \( i \leftarrow 0 \)
(4--) \( j \leftarrow 0 \)
(5--) \( \text{while} \ (i + j < n) \{ \)
(5.1) \( x \leftarrow x \times i \)
(5.2) \( x \leftarrow x + (j + 1) \)
(5.3) \( i \leftarrow i + 1 \)
(5.4) \( j \leftarrow j + 2 \)
(5--) \}
(6--) \( i \leftarrow 0 \)
(7--) \( j \leftarrow 0 \)
(8--) \( \text{while} \ (i + j < n) \{ \)
(8.1) \( y \leftarrow y \times j \)
(8.2) \( y \leftarrow y + (i + 2) \)
(8.3) \( i \leftarrow i + 1 \)
(8.4) \( j \leftarrow j + 2 \)
(8--) \}

post = 0 < x + y

Syntax: \text{fission} \ c \ @ \ m, \ n. \ Equivalent \ to \ \text{fission} \ c!1 \ @ \ m, \ n.

Syntax: \text{fission}(1) \cdots | \text{fission}(2) \cdots. \ The \ PRHL \ versions \ of \ the \ above \ variants, \ working \ on \ the \ designated \ program.

\[ \text{fusion} \]

Syntax: \text{fusion} \ c!l \ @ \ m, \ n. \ HL \ statement \ judgement \ version. \ Fails \ unless \ 0 \leq l \ and \ 0 \leq m \ and \ 0 \leq n \ and \ the \ cth \ statement \ of \ the \ program \ is \ a \ \text{while} \ \text{statement}, \ and \ there \ are \ at \ least \ l \ statements \ right \ before \ the \ \text{while} \ \text{statement}, \ at \ its \ level, \ and \ the \ part \ of \ the \ program \ beginning \ from \ the \ l \ statements \ before \ the \ while \ loop \ may \ be \ uniquely \ matched \ against

\[
\begin{align*}
\text{s}_1 & \ \text{while} \ (e) \ \{ \ s_2 \ s_4 \ \} \\
\text{s}_1 & \ \text{while} \ (e) \ \{ \ s_3 \ s_4 \ \}
\end{align*}
\]

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 : \( \text{M.f} \)

\[
\begin{align*}
\text{pre} &= 0 < n \\
(1--) & \ x \leftarrow 0 \\
(2--) & \ y \leftarrow 0 \\
(3--) & \ i \leftarrow 0 \\
(4--) & \ j \leftarrow 0 \\
(5--) & \textbf{while } (i + j < n) \{ \\
(5.1) & \ x \leftarrow x * i \\
(5.2) & \ x \leftarrow x + (j + 1) \\
(5.3) & \ i \leftarrow i + 1 \\
(5.4) & \ j \leftarrow j + 2 \\
(5--) & \}
\end{align*}
\]
\[
\begin{align*}
(6--) & \ i \leftarrow 0 \\
(7--) & \ j \leftarrow 0 \\
(8--) & \textbf{while } (i + j < n) \{ \\
(8.1) & \ y \leftarrow y * j \\
(8.2) & \ y \leftarrow y + (i + 2) \\
(8.3) & \ i \leftarrow i + 1 \\
(8.4) & \ j \leftarrow j + 2 \\
(8--) & \}
\end{align*}
\]

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

then running

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

produces the goal

Type variables: <none>

Context : \( \text{M.f} \)

\[
\begin{align*}
\text{pre} &= 0 < n \\
(1--) & \ x \leftarrow 0 \\
(2--) & \ y \leftarrow 0 \\
(3--) & \ i \leftarrow 0 \\
(4--) & \ j \leftarrow 0 \\
(5--) & \textbf{while } (i + j < n) \{ \\
(5.1) & \ x \leftarrow x * i \\
(5.2) & \ x \leftarrow x + (j + 1) \\
(5.3) & \ y \leftarrow y * j
\end{align*}
\]
(5.4) \( y \leftarrow y + (i + 2) \)
(5.5) \( i \leftarrow i + 1 \)
(5.6) \( j \leftarrow j + 2 \)
(5--) \}

post = 0 < x + y

Syntax: `fusion c \& m, n`. Equivalent to `fusion c!1 \& m, n`.

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

© `alias`

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

Syntax: `alias c`. Equivalent to `alias c with x`.

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 \( x \).

Syntax: `alias\{1\} \cdots \| alias\{2\} \cdots`. The pRHL versions of the preceding forms, where the aliasing is done in the designated program.

For example, if the current goal is

```
Type variables: <none>
Context : M.f
pre = true

(1) x <= 10
(2) (y, z) <= (x + 1, 6)

post = y + z = 17
```

then running

```
alias 2 with u.
```

produces the goal

```
Type variables: <none>
Context : M.f
```
pre = true

(1) x <- 10
(2) w <- (x + 1, 6)
(3) ⟨y, z⟩ <- w

post = y + z = 17

from which running

alias 3 u = w.‘1 + 7.

produces the goal

Type variables: <none>

Context : M.f

pre = true

(1) x <- 10
(2) w <- (x + 1, 6)
(3) u <- w.‘1 + 7
(4) ⟨y, z⟩ <- w

post = y + z = 17

from which running

alias 3.

produces the goal

Type variables: <none>

Context : M.f

pre = true

(1) x <- 10
(2) w <- (x + 1, 6)
(3) x₀ <- w.‘1 + 7
(4) u <- x₀
(5) ⟨y, z⟩ <- w

post = y + z = 17

\textbf{cfold}

Syntax: \texttt{cfold c ! m}. Fails unless \( m \geq 0 \). If the goal’s conclusion is an HL statement judgement in which statement \( c \) of the judgement’s program is an ordinary assignment statement in which constant values are assigned to local identifiers, and the following statement block of length \( m \) does not write any of those identifiers, then replace all occurrences of the assigned identifiers in that statement block by the constants assigned to them, and move the assignment statement to after the modified statement block.

For example, if the current goal is

Type variables: <none>
then running
\[ \text{cfold} \ 1 \ ! \ 1. \]
produces the goal

Type variables: <none>

from which running
\[ \text{cfold} \ 2. \]
produces the goal

Type variables: <none>

from which running
\[ \text{cfold} \ 1. \]
produces the goal

Type variables: <none>
pre = true
(1) z <- 1 + 1 + 1 + 2
(2) w <- 1 - z
(3) x <- 1
(4) y <- 1 + 1
post = w = -4

from which running
   cfold 1.
produces the goal
   Type variables: <none>
   Context : M.f
   pre = true
   (1) w <- 1 - (1 + 1 + 1 + 2)
   (2) x <- 1
   (3) y <- 1 + 1
   (4) z <- 1 + 1 + 1 + 2
   post = w = -4

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

Syntax: \texttt{cfold}\{1\} c | \texttt{cfold}\{2\} c. Like the general cases, but where \( m \) is set so as to be the number of statements after the assignment statement.

\( \odot \) \texttt{kill}

Syntax: \texttt{kill} c ! m. Fails unless \( m \geq 0 \). If the goal’s conclusion is an HL statement judgement whose program has a statement block starting at position \( c \) and having length \( m \) (when \( m = 0 \), this block is empty), and the variables written by this statement block aren’t used in the judgement’s postcondition or read by the rest of the program, then reduce the goal to two subgoals.

- One whose conclusion is a PHL statement judgement whose pre- and postconditions are true, whose program is the statement block, and whose bound part is \( 1 \% x \).

- 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

\texttt{kill 2 ! 2.}

produces the goals

Type variables: <none>

Context : M.f
Bound : \([=1] 1\%r\)

pre = true

(1) M.x \leftarrow 0
(2) M.x \leftarrow M.x + 1

post = true

and

Type variables: <none>

Context : M.f

pre = M.y = 3

(1) M.y \leftarrow M.y + 1
(2) M.y \leftarrow M.y + 1

post = M.y = 5

\textbf{Syntax:} \texttt{kill\{} c \mid m \texttt{ | kill\{} 2 c \mid m \texttt{.} Like the } \texttt{HL} \texttt{ case but for } \texttt{pRHL} \texttt{ judgements, where the statement block to be killed is in the designated program.}

For example, if the current goal is

Type variables: <none>

\&1 (\textit{left}) : M.f [programs are \textit{in sync}]
\&2 (\textit{right}) : M.f

pre = \texttt{=} M.y \texttt{ /\ M.y\{1\} = 3}

(1) M.y \leftarrow M.y + 1
(2) M.x \leftarrow 0
(3) M.x \leftarrow M.x + 1
(4) M.y \leftarrow M.y + 1

post = \texttt{=} M.y \texttt{ /\ M.y\{1\} = 5}

then running

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

produces the goals

Type variables: <none>

Context : M.f
CHAPTER 3. TACTICS

Bound : \text{[=} 1/2r

pre = true

(1) M.x <- 0
(2) M.x <- M.x + 1

post = true

and

Type variables: <none>

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

pre = \{M.y\} \& \ M.y{1} = 3

M.y <- M.y + 1 \quad (1) \quad M.y <- M.y + 1
M.x <- 0 \quad (2) \quad M.y <- M.y + 1
M.x <- M.x + 1 \quad (3)
M.y <- M.y + 1 \quad (4)

post = \{M.y\} \& \ M.y{1} = 5

Syntax: \textit{kill } c \mid \textit{kill}\{1\} c \mid \textit{kill}\{2\} c. Like the general cases, but with \(m = 1\).

Syntax: \textit{kill } c \mid \textit{kill}\{1\} c \mid \textit{kill}\{2\} c \mid \ast. Like the general cases, but with \(m\) set so that the statement block to be killed is the rest of the current level of the (designated) program.

3.4.3 Tactics for Reasoning about Specifications

\textit{symmetry}

Syntax: \textit{symmetry}. If the goal’s conclusion is a \texttt{pRHL} (statement) judgement, swap the two programs, transforming the pre- and postconditions by swapping the memories they refer to.

For example, if the current goal is

Type variables: <none>

\begin{align*}
\text{pre} &= x\{1\} = y\{2\} \& x\{2\} = - y\{1\} \\
M.f &\equiv N.g \\
\text{post} &= 1 - \text{res}\{1\} = \text{res}\{2\}
\end{align*}

then running

\textit{symmetry},

produces the goal

Type variables: <none>

\begin{align*}
\text{pre} &= x\{2\} = y\{1\} \& x\{1\} = - y\{2\} \\
N.g &\equiv M.f \\
\text{post} &= 1 - \text{res}\{2\} = \text{res}\{1\}
\end{align*}
And, if the current goal is

\[
\text{Type variables: } \langle \text{none} \rangle \\
\&1 \ \text{(left)} : M.f \\
\&2 \ \text{(right)} : N.g \\
\text{pre} = \\
(x{1}, y{1}).\cdot 1 = (x{2}, y{2}).\cdot 2 \\
(x{2}, y{2}).\cdot 1 = -(x{1}, y{1}).\cdot 2 \\
z \leftarrow x - y \\
\text{(1) } z \leftarrow x + y \\
\text{post} = 1 - (z{1} + 1) = -z{2}
\]

then running

\text{symmetry.}

produces the goal

\[
\text{Type variables: } \langle \text{none} \rangle \\
\&1 \ \text{(left)} : N.g \\
\&2 \ \text{(right)} : M.f \\
\text{pre} = \\
(x{2}, y{2}).\cdot 1 = (x{1}, y{1}).\cdot 2 \\
(x{1}, y{1}).\cdot 1 = -(x{2}, y{2}).\cdot 2 \\
z \leftarrow x + y \\
\text{(1) } z \leftarrow x - y \\
\text{post} = 1 - (z{2} + 1) = -z{1}
\]

\(\textsf{transitivity}\)

\textbf{Syntax: transitivity} \(N.r \ (P_1 \implies Q_1) \ (P_2 \implies Q_2)\). Reduces a goal whose conclusion is a \(rRHL\) judgement (not statement judgement)

\[
equiv[M_1.p_1 - M_2.p_2 : P \implies Q]
\]

to goals whose conclusions are

- \(\equiv[M_1.p_1 - N.r : P_1 \implies Q_1]\) and
- \(\equiv[N.r - M_2.p_2 : P_2 \implies Q_2]\),

preceded by two auxiliary goals. The tactic fails if the \(P_i\) and \(Q_i\) aren’t compatible with these left and right programs. The conclusion of the first auxiliary goal checks that \(P\) implies the conjunction of \(P_1\) and \(P_2\), where each corresponding pair of \&2 memory references in \(P_1\) and \&1 reference in \(P_2\) is existentially quantified. And the conclusion of the second auxiliary goal checks that the conjunction of \(Q_1\) and \(Q_2\) implies \(Q\), where each corresponding pair of \&2 references in \(Q_1\) and \&1 references in \(Q_2\) are universally quantified.

For example, consider the modules

\texttt{module } M = 
\texttt{ proc f(n : int, m : int) : int = }
\texttt{ var i, x : int; }
\texttt{ i <- 0; x <- 0;
CHAPTER 3. TACTICS

while (i < n) {
    x <- x + m; i <- i + 1;
} return x;
}

module N = {
    proc g(n : int, m : int) : int = {
        var j, y : int;
        j <- 0; y <- 0;
        while (j < n) {
            y <- y + m; j <- j + 1;
        }
        return y;
    }
}

module R = {
    proc h(n : int, m : int) : int = {
        return n * m;
    }
}

If the current goal is

Type variables: <none>

\[\text{pre} = 0 \leq n(1) \land 0 \leq n(2) \land n(1) \cdot m(1) = n(2) \cdot m(2)\]

\[M.f \rightarrow N.g\]

\[\text{post} = \{\text{res}\}\]

then running

\[\text{transitivity}\]

\[R.h\]

\[\begin{align*}
0 \leq n(1) \land (n \cdot m)(1) &= (n \cdot m)(2) 
\Rightarrow &\implies \{\text{res}\} \\
0 \leq n(2) \land (n \cdot m)(1) &= (n \cdot m)(2) 
\Rightarrow &\implies \{\text{res}\}
\end{align*}\]

produces the four goals

Type variables: <none>

\[\text{forall} \ &1 \ &2, \\
0 \leq n(1) \land 0 \leq n(2) \land n(1) \cdot m(1) = n(2) \cdot m(2) \Rightarrow \exists (\text{arg} : \text{int} \times \text{int}),
\]

\[\begin{align*}
(0 \leq n(1) \land n(1) \cdot m(1) &= \text{arg} \cdot 1 \cdot \text{arg} \cdot 2) 
\land 0 \leq n(2) \land \text{arg} \cdot 1 \cdot \text{arg} \cdot 2 &= n(2) \cdot m(2)
\end{align*}\]

and

Type variables: <none>

\[\text{forall} \ &1 \ &m \ &2, \ \text{res}(1) = \text{res}(m) \Rightarrow \text{res}(m) = \text{res}(2) \Rightarrow \{\text{res}\}\]

and

Type variables: <none>
CHAPTER 3. TACTICS

pre = 0 <= n{1} /\ n{1} * m{1} = n{2} * m{2}

M.f - R.h

post = ={res}

and

Type variables: <none>

pre = 0 <= n{2} /\ n{1} * m{1} = n{2} * m{2}

R.h - N.g

post = ={res}

Syntax: transitivity(i) {s} (P1 ==> Q1) (P2 ==> Q2).
Reduces a goal whose conclusion is a PRHL statement judgement with precondition P, postcondition Q, left program (statement sequence) s1 and right program s2 to goals whose conclusions are PRHL statement judgements:

• equiv[s1 - s : P1 ==> Q1] and
• equiv[s - s2 : P2 ==> Q2],
preceded by two auxiliary goals. If the side i = 1, then the statement sequence s may only use variables and unqualified procedures of the left program; when i = 2, it may only use variables and unqualified procedures of the right program. The tactic fails if the P and Q aren’t compatible with these left and right programs. The conclusion of the first auxiliary goal checks that P implies the conjunction of P1 and P2, where each corresponding pair of &2 memory references in P1 and &1 reference in P2 is existentially quantified. And the conclusion of the second auxiliary goal checks that the conjunction of Q1 and Q2 implies Q, where each corresponding pair of &2 memory references in Q1 and &1 references in Q2 are universally quantified.

Consider the modules M, N and R of the preceding case. If the current goal is

Type variables: <none>

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

pre =
  j{2} = 0 /\ y{2} = 0 /\ i{1} = 0 /\ x{1} = 0 /\ 0 <= n{1} /\ 0 <= n{2} /\ n{1} * m{1} = n{2} * m{2}

while (i < n) {
  x <- x + m
  i <- i + 1
}

while (j < n) {
  y <- y + m
  j <- j + 1
}

post = x{1} = y{2}

then running

transitivity{1}
{x <- n * m;}

(0 <= n{1} /\ i{1} = 0 /\ x{1} = 0 /\ (n * m){1} = (n * m){2} ==>

pre = 0 <= n{1} /\ n{1} * m{1} = n{2} * m{2}

M.f - R.h

post = ={res}
produces the four goals

\[\forall \alpha, \beta, \gamma, \delta, \\alpha = \beta \rightarrow \gamma = \delta\]

and

\[\forall \alpha, \beta, \gamma, \delta, \\alpha = \beta \rightarrow \gamma = \delta \rightarrow \alpha = \gamma\]

And, if the current goal is
Type variables: <none>

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

pre =
\[ j(2) = 0 \land y(2) = 0 \land i(1) = 0 \land x(1) = 0 \land 0 \leq n(1) \land 0 \leq n(2) \land n(1) \ast m(1) = n(2) \ast m(2) \]

\textbf{while} (i < n) {
\hspace{1em} (1--) \textbf{while} (j < n) {
\hspace{2em} x \leftarrow x + m \hspace{1em} (1.1) \hspace{1em} y \leftarrow y + m
\hspace{2em} i \leftarrow i + 1 \hspace{1em} (1.2) \hspace{1em} j \leftarrow j + 1
\}
\}

post = x(1) = y(2)

then running

\textbf{transitivity}(2): \{ y \leftarrow n \ast m ; \}
\hspace{1em} (0 \leq n(1) \land i(1) = 0 \land x(1) = 0 \land (n \ast m)(1) = (n \ast m)(2) \implies x(1) = y(2))
\hspace{1em} (0 \leq n(2) \land j(2) = 0 \land y(2) = 0 \land (n \ast m)(1) = (n \ast m)(2) \implies y(2) = y(2))

produces the four goals

Type variables: <none>

\textbf{forall} \&1 \&2, \hspace{1em} j(2) = 0 \land y(2) = 0 \land i(1) = 0 \land x(1) = 0 \land 0 \leq n(1) \land 0 \leq n(2) \land n(1) \ast m(1) = n(2) \ast m(2) \implies \exists (m n : \text{int}), \hspace{1em} (0 \leq n(1) \land i(1) = 0 \land x(1) = 0 \land (n \ast m)(1) = (n \ast m) \land (0 \leq n(2) \land j(2) = 0 \land y(2) = 0 \land n \ast m = n(2) \ast m(2))

and

Type variables: <none>

\textbf{forall} \&1 \&m \&2, \hspace{1em} x(1) = y(m) \implies y(m) = y(2) \implies x(1) = y(2)

and

Type variables: <none>

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

pre =
\[ 0 \leq n(1) \land i(1) = 0 \land x(1) = 0 \land n(1) \ast m(1) = n(2) \ast m(2) \]

\textbf{while} (i < n) {
\hspace{1em} (1--) y \leftarrow n \ast m
CHAPTER 3. TACTICS

\[ x \leftarrow x + m \]  
(1.1)

\[ i \leftarrow i + 1 \]  
(1.2)

\}

(1--)

post = x{1} = y{2}

and

Type variables: <none>

\&1 (left ) : N.g

\&2 (right ) : N.g

pre =

\[ 0 \leq n{2} \land j{2} = 0 \land y{2} = 0 \land n{1} \times m{1} = n{2} \times m{2} \]

\[
y \leftarrow n \times man{2}

(1--) while \( j < n \) {

(1.1) \[ y \leftarrow y + m \]  

(1.2) \[ j \leftarrow j + 1 \]

(1--)

}\}

post = ={y}

\[ \text{conseq} \]

\text{Syntax: } \text{conseq} \ (\_ : P \Rightarrow Q).

If the goal’s conclusion is a \( \text{pRHL} \) or \( \text{HL} \) judgement or statement judgement, weaken the conclusion’s precondition to \( P \), and strengthen its postcondition to \( Q \), generating initial auxiliary subgoals checking that this reduction is sound. Fails if \( P \) and \( Q \) aren’t compatible with the judgement type. The conclusion of the first auxiliary subgoal checks that the precondition \( P' \) of the original goal’s conclusion implies \( P \). The conclusion of the second auxiliary subgoal checks that \( Q \) implies the postcondition \( Q' \) of the original goal’s conclusion, except that any memory references to variables that may be modified by the conclusion’s program(s) are universally quantified in \( Q \) and \( Q' \), and \( P \) is also included as an assumption.

\( P \) or \( Q \) may be replaced by \( \_ \), in which case the pre- or postcondition is left unchanged. When a pre- or postcondition is unchanged, the corresponding auxiliary subgoal isn’t generated (as its proof would be trivial). When the goal’s conclusion is a judgement (not a statement judgement), a proof term whose conclusion is a judgement on the procedure(s) of the original goal’s conclusion may be supplied as the argument to \text{conseq}, in which case \( P \) and \( Q \) are taken to be the pre- and postconditions of this judgement, and only the auxiliary subgoals are generated.

For example, if the current goal is

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

\&1 (left ) : M.f

\&2 (right ) : N.f

pre =

\[ =z \land \]

M.x{1} = N.x{2} \land

M.y{1} = N.y{2} \land z{1} < M.x{1} \land z{1} < M.y{1}

\[
if \ (z \leq M.x \land z \leq M.y) \ { \ (1--) \ \ if \ (z \leq N.x) \ { \\
M.x \leftarrow M.x + 1 \ 
(1.1) \ \ N.x \leftarrow N.x - 1 \ 
(1--) \ }
\}
post = M.y{1} = N.y{2} \land \ |M.x{1} - N.x{2}| \leq 2 \]
then running

```
conseq (_ : ={z} \ M.x{1} = N.x{2} \ M.y{1} = N.y{2} \ z{1} <= M.x{1} \ z{1} <= M.y{1} ==>_ _).
```

produces the goals

Type variables: <none>

```
forall &1 &2,
  ={z} \ M.x{1} = N.x{2} \ M.y{1} = N.y{2} \ z{1} <= M.x{1} \ z{1} <= M.y{1} =>
  ={z} \ M.x{1} = N.x{2} \ M.y{1} = N.y{2} \ z{1} <= M.x{1} \ z{1} <= M.y{1}
```

and

Type variables: <none>

```
&1 (left ) : M.f
&2 (right) : N.f
pre =
  ={z} \ M.x{1} = N.x{2} \ M.y{1} = N.y{2} \ z{1} <= M.x{1} \ z{1} <= M.y{1}
  if (z <= M.x \ z <= M.y) { 1--  if (z <= N.x) {
    M.x <= M.x + 1 (1.1) N.x <= N.x - 1
  } (1-- ) }
post = M.y{1} = N.y{2} \ |M.x{1} - N.x{2}| <= 2
```

Continuing from the last of these goals, running

```
conseq (_ : _ ==>_ |M.x{1} - N.x{2}| <= 2).
```

produces the goals

Type variables: <none>

```
forall &1 &2,
  ={z} \ M.x{1} = N.x{2} \ M.y{1} = N.y{2} \ z{1} <= M.x{1} \ z{1} <= M.y{1} =>
  forall (x_L x_R : int),
  `|x_L - x_R| <= 2 => M.y{1} = N.y{2} \ `|x_L - x_R| <= 2
```

and

Type variables: <none>

```
&1 (left ) : M.f
&2 (right) : N.f
pre =
  ={z} \ 
```
Continuing from the last of these goals, running

\[ \text{conseq} \left( \_ : \right. \\
\quad =\{z\} \land M.x{1} = N.x{2} \land z{1} \leq N.x{2} \Rightarrow M.x{1} = N.x{2} \lor M.x{1} = N.x{2} + 2 \).
\]

produces the goals

Type variables: <none>

\[ \forall \&1 \&2, \quad =\{z\} \land M.x{1} = N.x{2} \land M.y{1} = N.y{2} \land z{1} \leq M.x{1} \land z{1} \leq M.y{1} \Rightarrow =\{z\} \land M.x{1} = N.x{2} \land z{1} \leq N.x{2} \]

and

Type variables: <none>

\[ \forall \&1 \&2, \quad =\{z\} \land M.x{1} = N.x{2} \land M.y{1} = N.y{2} \land z{1} \leq M.x{1} \land z{1} \leq M.y{1} \Rightarrow \forall (x_L x_R : \text{int}), \quad x_L = x_R \lor x_L = x_R + 2 \Rightarrow =\{x_L - x_R\} \leq 2 \]

and

Type variables: <none>

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

\[ \text{pre} = =\{z\} \land M.x{1} = N.x{2} \land z{1} \leq N.x{2} \]

\[ \text{if} (z \leq M.x \land z \leq N.y) \left\{ \text{(1--)} \quad \text{if} (z \leq N.x) \left\{ \\
M.x \leftarrow M.x + 1 \quad \text{(1.1)} \quad N.x \leftarrow N.x - 1 \\
\right. \right. \}
\]

\[ \text{post} = =\{M.x{1} - N.x{2}\} \leq 2 \]

If the current goal is

Type variables: <none>

\[ \text{pre} = =\{z\} \land M.x{1} = N.x{2} \land M.y{1} = N.y{2} \land z{1} < M.x{1} \land z{1} < M.y{1} \]
CHAPTER 3. TACTICS

\[ \text{M.f} \sim \text{N.f} \]

\[ \text{post} = M.y(1) = N.y(2) \land \lnot |M.x(1) - N.x(2)| \leq 2 \]

then running

\[
\text{conseq} \quad (_ : \quad =z \quad \land \quad M.x(1) = N.x(2) \land M.y(1) = N.y(2) \land z(1) \leq M.x(1) \land z(1) \leq M.y(1) \implies _) \]

produces the goals

Type variables: <none>

\[
\forall &1 \quad &2, \quad =z \quad \land \quad M.x(1) = N.x(2) \land M.y(1) = N.y(2) \land z(1) \leq M.x(1) \land z(1) \leq M.y(1) \implies \forall (x_L \quad x_R : \text{int}), \quad \lnot |x_L - x_R| \leq 2 \implies M.y(1) = N.y(2) \land \lnot |x_L - x_R| \leq 2
\]

and

Type variables: <none>

\[
\text{pre} = \quad =z \quad \land \quad M.x(1) = N.x(2) \land M.y(1) = N.y(2) \land z(1) \leq M.x(1) \land z(1) \leq M.y(1)
\]

\[
\text{M.f} \sim \text{N.f}
\]

Continuing from the last of these goals, running

\[
\text{conseq} \quad (_ : \quad _ \implies \lnot |M.x(1) - N.x(2)| \leq 2).
\]

produces the goals

Type variables: <none>

\[
\forall &1 \quad &2, \quad =z \quad \land \quad M.x(1) = N.x(2) \land M.y(1) = N.y(2) \land z(1) \leq M.x(1) \land z(1) \leq M.y(1) \implies \forall (x_L \quad x_R : \text{int}), \quad \lnot |x_L - x_R| \leq 2 \implies M.y(1) = N.y(2) \land \lnot |x_L - x_R| \leq 2
\]

and

Type variables: <none>

\[
\text{pre} = \quad =z \quad \land \quad M.x(1) = N.x(2) \land M.y(1) = N.y(2) \land z(1) \leq M.x(1) \land z(1) \leq M.y(1)
\]

\[
\text{M.f} \sim \text{N.f}
\]
Continuing from the last of these goals, running

\[
\text{conseq} \quad (\exists z. \ (M.x(1) = N.x(2) \land z(1) \leq N.x(2)) \implies M.x(1) = N.x(2) \lor M.x(1) = N.x(2) + 2).
\]

produces the goals

\[
\text{forall } &1 &2, \\
\quad \{z\} \land M.x(1) = N.x(2) \land M.y(1) = N.y(2) \land z(1) \leq M.x(1) \land z(1) \leq M.y(1) \implies \{z\} \land M.x(1) = N.x(2) \\
\quad \forall (x_L x_R : \text{int}), \\
\quad x_L = x_R \lor x_L = x_R + 2 \implies \lvert x_L - x_R \rvert \leq 2
\]

and

\[
\text{forall } &1 &2, \\
\quad \{z\} \land M.x(1) = N.x(2) \land M.y(1) = N.y(2) \land z(1) < M.x(1) \land z(1) < M.y(1) \implies \{z\} \land M.x(1) = N.x(2) \\
\quad \lvert M.x(1) - N.x(2) \rvert \leq 2
\]

Given lemma

\[
\text{lemma } M.N.3 : \\
\equiv \{z\} \land M.x(1) = N.x(2) \land z(1) \leq N.x(2) \implies \{z\} \land M.x(1) = N.x(2) \lor M.x(1) = N.x(2) + 2.
\]

if the current goal is

\[
\text{pre} = \{z\} \land M.x(1) = N.x(2) \land M.y(1) = N.y(2) \land z(1) < M.x(1) \land z(1) < M.y(1) \\
\quad \lvert M.x(1) - N.x(2) \rvert \leq 2
\]
then running

\texttt{conseq \_ : x' = M.x /\ z <= M.y ==> x' = M.x - 1).}

produces the goals

Type variables: <none>

\texttt{x': int}

\texttt{forall \&hr,}

\begin{align*}
&x' = M.x(\&hr) /\ z(\&hr) < M.x(\&hr) /\ z(\&hr) < M.y(\&hr) => \\
&x' = M.x(\&hr) /\ z(\&hr) <= M.y(\&hr)
\end{align*}

and

Type variables: <none>

\texttt{x': int}
```plaintext
forall &hr,
  x' = M.x{hr} \ z{hr} < M.x{hr} \ z{hr} < M.y{hr} =>
for all (x : int), x' = x - 1 => x' < x
```

and

Type variables: <none>

x': int

Context : M.f

pre = x' = M.x \ z <= M.y

(1--) if (z <= M.x \/ z <= M.y) {
(1.1) M.x <- M.x + 1
(1--) }

post = x' = M.x - 1

If the current goal is

Type variables: <none>

x': int

pre = x' = M.x \ z < M.x \ z < M.y

M.f

post = x' < M.x

then running

```plaintext
conseq (_ : x' = M.x \ z <= M.y ==> x' = M.x - 1).
```

produces the goals

Type variables: <none>

x': int

```plaintext
forall &hr,
  x' = M.x{hr} \ z{hr} < M.x{hr} \ z{hr} < M.y{hr} =>
  x' = M.x{hr} \ z{hr} <= M.y{hr}
```

and

Type variables: <none>

```plaintext
forall &hr,
  x' = M.x{hr} \ z{hr} < M.x{hr} \ z{hr} < M.y{hr} =>
  forall (x : int), x' = x - 1 => x' < x
```

and

Type variables: <none>

x': int
pre = \( x' = M.x \land z \leq M.y \)

\( M.f \)

post = \( x' = M.x - 1 \)

Given lemma

\begin{verbatim}
lemma M_3 (x' : int) :
  hoare[M.f : x' = M.x \land z \leq M.y ==> x' = M.x - 1].
\end{verbatim}

if the current goal is

Type variables: <none>

\( x' : \text{int} \)

pre = \( x' = M.x \land z < M.x \land z < M.y \)

\( M.f \)

post = \( x' < M.x \)

then running

\begin{verbatim}
conseq (M_3 x').
\end{verbatim}

produces the goals

Type variables: <none>

\( x' : \text{int} \)

forall \( \sigma, x' = M.x(\sigma) \land z(\sigma) < M.x(\sigma) \land z(\sigma) < M.y(\sigma) \rightarrow x' = M.x(\sigma) \land z(\sigma) \leq M.y(\sigma) \)

and

Type variables: <none>

\( x' : \text{int} \)

forall \( \sigma, x' = M.x(\sigma) \land z(\sigma) < M.x(\sigma) \land z(\sigma) < M.y(\sigma) \rightarrow\)

\begin{verbatim}
forall (x : \text{int}), x' = x - 1 \rightarrow x' < x
\end{verbatim}

Syntax: \begin{verbatim}
conseq (\_ : P \implies Q) (\_ : P_1 \implies Q_1) (\_ : P_2 \implies Q_2).
\end{verbatim}

This form only applies to pRHL judgements and statement judgements, reducing the goal’s conclusion to

- an HL (statement) judgement for the left program with precondition \( P_1 \) and postcondition \( Q_1 \);
- an HL (statement) judgement for the right program with precondition \( P_2 \) and postcondition \( Q_2 \); and
- a pRHL (statement) judgement whose precondition is \( P \), postcondition is \( Q \), and programs are as in the original judgement;

As before, auxiliary goals are generated whose conclusions check the validity of the reduction: that the original conclusion’s precondition \( P' \) implies the conjunction of \( P, P_1 \) and \( P_2 \); and that
the conjunction of \( Q, Q_1 \) and \( Q_2 \) implies the original conclusion’s postcondition \( Q' \). One of the HL specifications may be replaced by \(_\_\), which is equivalent to \texttt{conseq} \(_\_ : \text{true} \implies \text{true} \). And in the case of a \texttt{pRHL} judgement (not a statement judgement), proof terms may be used as the arguments to \texttt{conseq}.

For example, if the current goal is

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

\[
\text{
\&1 (left) : M.f \\
\&2 (right) : N.f \\
pre = (\{b\} \land M.x{1} \mod 2 + N.y{2} \mod 2 = 0 \\
M.x \leftarrow M.x + 2 \land b' < 0 \{0,1\} \land b \leftarrow b \land b' \\
post = (\{b\} \land M.x{1} \mod 2 + N.y{2} \mod 2 = 0}
\]

then running

\[
\text{\texttt{conseq (_ : \{b\} \implies \{b\})}}
\]

\[
\text{(_ : M.x \mod 2 = 0 \implies M.x \mod 2 = 0) \\
(_ : N.y \mod 2 = 0 \implies N.y \mod 2 = 0).}
\]

produces the goals

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

\[
\text{\forall \&1 \&2,} \\
\text{\{b\} \land M.x{1} \mod 2 + N.y{2} \mod 2 = 0 \implies} \\
\text{\{b\} \land M.x{1} \mod 2 = 0 \land N.y{2} \mod 2 = 0}
\]

and

\[
\text{\forall \&1 \&2,} \\
\text{\{b\} \land M.x{1} \mod 2 + N.y{2} \mod 2 = 0 \implies} \\
\text{\forall \langle x_L : \text{int} \rangle \langle b_L : \text{bool} \rangle \langle y_R : \text{int} \rangle \langle b_R : \text{bool} \rangle,} \\
\text{\{b\} \land b_L \land x_L \mod 2 = 0 \land y_R \mod 2 = 0 \implies} \\
\text{\{b\} \land b_R \land x_L \mod 2 + y_R \mod 2 = 0}
\]

and

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

\[
\text{pre = M.x \mod 2 = 0} \\
(1) \text{M.x \leftarrow M.x + 2} \\
(2) \text{b' < 0 \{0,1\}} \\
(3) \text{b \leftarrow b \land b'} \\
\text{post = M.x \mod 2 = 0}
\]
CHAPTER 3. TACTICS

Type variables: <none>

Context : N.f

pre = N.y \%\% 2 = 0

(1) N.y \leftarrow N.y - 2
(2) b' <\{0,1\}
(3) b \leftarrow b \setminus b'

post = N.y \%\% 2 = 0

and

Type variables: <none>

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

pre = =\{b\}

M.x \leftarrow M.x + 2
b' <\{0,1\}
b \leftarrow b \setminus b'

post = =\{b\}

If the current goal is

Type variables: <none>

pre = =\{b\} \setminus M.x\{1\} \%\% 2 + N.y\{2\} \%\% 2 = 0

M.f \sim N.f

post = =\{res\} \setminus M.x\{1\} \%\% 2 + N.y\{2\} \%\% 2 = 0

then running

\text{conseq} (_ : =\{b\} \implies =\{res\})

(_ : M.x \%\% 2 = 0 \implies M.x \%\% 2 = 0)

(_ : N.y \%\% 2 = 0 \implies N.y \%\% 2 = 0).

produces the goals

Type variables: <none>

\text{forall} &1 &2,
\quad =\{b\} \setminus M.x\{1\} \%\% 2 + N.y\{2\} \%\% 2 = 0 =>
\quad =\{b\} \setminus M.x\{1\} \%\% 2 = 0 \setminus N.y\{2\} \%\% 2 = 0

and

Type variables: <none>

\text{forall} &1 &2,
\quad =\{b\} \setminus M.x\{1\} \%\% 2 + N.y\{2\} \%\% 2 = 0 =>
forall (result_L result_R : bool) (x_L y_R : int), result_L = result_R \( \land \) x_L \( \%\% \) 2 = 0 \( \land \) y_R \( \%\% \) 2 = 0 => result_L = result_R \( \land \) x_L \( \%\% \) 2 + y_R \( \%\% \) 2 = 0

and

Type variables: <none>

pre = M.x \( \%\% \) 2 = 0
M.f
post = M.x \( \%\% \) 2 = 0

and

Type variables: <none>

pre = N.y \( \%\% \) 2 = 0
N.f
post = N.y \( \%\% \) 2 = 0

and

Type variables: <none>

pre = ={b}
M.f ~ N.f
post = ={res}

And given lemmas

lemma M_N : equiv[M.f ~ N.f : ={b} => ={res}].

and

lemma M : hoare[M.f : M.x \( \%\% \) 2 = 0 => M.x \( \%\% \) 2 = 0].

and

lemma N : hoare[N.f : N.y \( \%\% \) 2 = 0 => N.y \( \%\% \) 2 = 0].

if the current goal is

pre = ={b} /\ M.x{1} \( \%\% \) 2 + N.y{2} \( \%\% \) 2 = 0
M.f ~ N.f
post = ={res} /\ M.x{1} \( \%\% \) 2 + N.y{2} \( \%\% \) 2 = 0

then running

conseq M_N M N.
produces the goals

\[
\text{forall } \&1 \&2, \\
=\{b\} \land M.x{1} \% 2 + N.y{2} \% 2 = 0 \Rightarrow \\
=\{b\} \land M.x{1} \% 2 = 0 \land N.y{2} \% 2 = 0
\]

and

\[
\text{forall } \&1 \&2, \\
=\{b\} \land M.x{1} \% 2 + N.y{2} \% 2 = 0 \Rightarrow \\
\text{forall } (\text{result}_L, \text{result}_R : \text{bool}) (x_L, y_R : \text{int}), \\
\text{result}_L = \text{result}_R \land x_L \% 2 = 0 \land y_R \% 2 = 0 \Rightarrow \text{result}_L = \text{result}_R \land x_L \% 2 + y_R \% 2 = 0
\]

### Syntax: case e

If the goal’s conclusion is a pRHL, HL or pHL statement judgement and e is well-typed in the goal’s context, split the goal into two goals:

- a first goal in which e is added as a conjunct to the conclusion’s precondition; and
- a second goal in which !e is added as a conjunct to the conclusion’s precondition.

For example, if the current goal is

\[
\text{forall } \&1 \&2, \\
=\{b\} \land M.x{1} \% 2 + N.y{2} \% 2 = 0 \Rightarrow \\
\text{forall } (\text{result}_L, \text{result}_R : \text{bool}) (x_L, y_R : \text{int}), \\
\text{result}_L = \text{result}_R \land x_L \% 2 = 0 \land y_R \% 2 = 0 \Rightarrow \text{result}_L = \text{result}_R \land x_L \% 2 + y_R \% 2 = 0
\]

then running

\[
\text{case } (x{1} < y{1}).
\]

produces the goals

\[
\text{forall } \&1 \&2, \\
=\{b\} \land x{1} < y{1} \Rightarrow \\
\text{forall } (\text{result}_L, \text{result}_R : \text{bool}) (x_L, y_R : \text{int}), \\
\text{result}_L = \text{result}_R \land x_L \% 2 = 0 \land y_R \% 2 = 0 \Rightarrow \text{result}_L = \text{result}_R \land x_L \% 2 + y_R \% 2 = 0
\]
\[ z \leftarrow y - x \quad (1.1) \quad z \leftarrow x - y \]
\[
\{ \text{else} \} \quad (1--) \quad \{ \text{else} \}
\]
\[
\{ z \leftarrow x - y \quad (1?1) \quad z \leftarrow y - x \}
\]
\[
\text{post} = \{ z \}
\]

and

Type variables: <none>

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

\[
\text{pre} = \{(x, y) / \backslash ! x \{1\} < y\{1\} \}
\]
\[
\text{if}(x < y) \{ \quad (1--) \quad \text{if}(y <= x) \{ \\
\quad z \leftarrow y - x \quad (1.1) \quad z \leftarrow x - y \\
\quad \} \quad (1--) \quad \} \quad \} \quad \} \quad \} \quad \}
\]
\[
\text{post} = \{ z \}
\]

And if the current goal is

Type variables: <none>

Context : M.f

\[
\text{pre} = \text{true}
\]
\[
(1--) \quad y \leftarrow -1
\]
\[
(2--) \quad \text{while} (0 < x) \{ \\
(2.1) \quad y \leftarrow y + 2 \\
(2.2) \quad x \leftarrow x - 1 \\
(2--) \}
\]
\[
\text{post} = y <> 0
\]

then running

\textbf{case} (0 < x).

produces the goals

Type variables: <none>

Context : M.f

\[
\text{pre} = 0 < x
\]
\[
(1--) \quad y \leftarrow -1
\]
\[
(2--) \quad \text{while} (0 < x) \{ \\
(2.1) \quad y \leftarrow y + 2 \\
(2.2) \quad x \leftarrow x - 1 \\
(2--) \}
\]
\[
\text{post} = y <> 0
\]
and

Type variables: <none>

Context : M.f

pre = ! 0 < x

(1--) y <- -1
(2--) while (0 < x) {
(2.1) y <- y + 2
(2.2) x <- x - 1
(2--) }

post = y <> 0

\(\circ\) byequiv

**Syntax:** byequiv \((_, P \Rightarrow Q)\). If the goal’s conclusion has the form

\[
\Pr[M_1.p_1(a_{1,1}, \ldots, a_{1,n_1}) \& &m_1 : E_1] = \Pr[M_2.p_2(a_{2,1}, \ldots, a_{2,n_2}) \& &m_2 : E_2],
\]

reduce the goal to three subgoals:

- One with conclusion \(\text{equiv}[M_1.p_1 - M_2.p_2 : P \Rightarrow Q]\);
- One whose conclusion says that \(P\) holds, where references to memories \&1 and \&2 have been replaced by \&m1 and \&m2, respectively, and references to the formal parameters of \(M_1.p_1\) and \(M_2.p_2\) have been replaced by their arguments;
- One whose conclusion says that \(Q\) implies that \(E_1(1) \iff E_2(2)\).

The argument to byequiv may be replaced by a proof term for \(\text{equiv}[M_1.p_1 - M_2.p_2 : P \Rightarrow Q]\), in which case the first subgoal isn’t generated. Furthermore, either or both of \(P\) and \(Q\) may be replaced by \_, asking that the pre- or postcondition be inferred. Supplying no argument to byequiv is the same as replacing both \(P\) and \(Q\) by \_. By default, inference of \(Q\) attempts to infer a conjunction of equalities implying \(E_1(1) \iff E_2(2)\). Passing the \[-eq\] option to byequiv takes \(Q\) to be \(E_1(1) \iff E_2(2)\).

The other variants of the tactic behave similarly with regards to the use of proof terms and specification inference.

For example, consider the module

```coffee
module M = {
  var x : int
  proc f(y : int) : int = {
    var z : int;
    z <$ [x .. y];
    return z;
  }
}.
```

If the current goal is

Type variables: <none>

y1: int
y2: int
&m1: memory
&m2: memory
\[ x_{eq}: \text{M.x}\{m1\} = \text{M.x}\{m2\} \]
\[ y_{eq}: y_1 = y_2 \]
\[ \Pr[\text{M.f}(y_1) \land \&m1 : \text{res} = 0] = \Pr[\text{M.f}(y_2) \land \&m2 : \text{res} = 0] \]

then running

\[
\text{byequiv} \ (_\downarrow : \{_\downarrow : \{\text{M.x, y}\} => \{\text{res}\}).
\]

produces the goals

Type variables: <none>

\[
y_1: \text{int} \\
y_2: \text{int} \\
\&m1: \text{memory} \\
\&m2: \text{memory} \\
x_{eq}: \text{M.x}\{m1\} = \text{M.x}\{m2\} \\
y_{eq}: y_1 = y_2 \\
\pre = \{\text{M.x, y}\} \\
\text{M.f} - \text{M.f} \\
\post = \{\text{res}\}
\]

and

Type variables: <none>

\[
y_1: \text{int} \\
y_2: \text{int} \\
\&m1: \text{memory} \\
\&m2: \text{memory} \\
x_{eq}: \text{M.x}\{m1\} = \text{M.x}\{m2\} \\
y_{eq}: y_1 = y_2 \\
\text{M.x}\{m1\} = \text{M.x}\{m2\} \land y_1 = y_2
\]

and

Type variables: <none>

\[
y_1: \text{int} \\
y_2: \text{int} \\
\&m1: \text{memory} \\
\&m2: \text{memory} \\
x_{eq}: \text{M.x}\{m1\} = \text{M.x}\{m2\} \\
y_{eq}: y_1 = y_2 \\
\forall\ &1\ &2, \{\text{res}\} \Rightarrow \text{res}\{1\} = 0 \Leftrightarrow \text{res}\{2\} = 0
\]

Given the lemma

\[
\text{lemma} \ M_M_f : \text{equiv}[\text{M.f} - \text{M.f} : \{\text{M.x, y}\} \Rightarrow \{\text{res}\}].
\]

if the current goal is

Type variables: <none>

\[
y_1: \text{int} \\
y_2: \text{int} \\
\&m1: \text{memory}
\]
CHAPTER 3. TACTICS

151

\[
\begin{align*}
&m_2: \text{memory} \\
&x_{\text{eq}}: M.x(m_1) = M.x(m_2) \\
&y_{\text{eq}}: y_1 = y_2
\end{align*}
\]

\[
\Pr[M.f(y_1) @ &m_1 : \text{res} = 0] = \Pr[M.f(y_2) @ &m_2 : \text{res} = 0]
\]

then running

\texttt{byequiv M\_M\_f}.

produces the goals

Type variables: <none>

\[
\begin{align*}
y_1: \text{int} \\
y_2: \text{int} \\
&m_1: \text{memory} \\
&m_2: \text{memory} \\
x_{\text{eq}}: M.x(m_1) = M.x(m_2) \\
y_{\text{eq}}: y_1 = y_2
\end{align*}
\]

\[
M.x(m_1) = M.x(m_2) \land y_1 = y_2
\]

and

Type variables: <none>

\[
\begin{align*}
y_1: \text{int} \\
y_2: \text{int} \\
&m_1: \text{memory} \\
&m_2: \text{memory} \\
x_{\text{eq}}: M.x(m_1) = M.x(m_2) \\
y_{\text{eq}}: y_1 = y_2
\end{align*}
\]

\[
\forall &1, &2, =\{\text{res}\} \Rightarrow \text{res}\{1\} = 0 \Rightarrow \text{res}\{2\} = 0
\]

And, if the current goal is

Type variables: <none>

\[
\begin{align*}
y_1: \text{int} \\
y_2: \text{int} \\
&m_1: \text{memory} \\
&m_2: \text{memory} \\
x_{\text{eq}}: M.x(m_1) = M.x(m_2) \\
y_{\text{eq}}: y_1 = y_2
\end{align*}
\]

\[
\Pr[M.f(y_1) @ &m_1 : \text{res} = 0] = \Pr[M.f(y_2) @ &m_2 : \text{res} = 0]
\]

then running

\texttt{byequiv}.

produces the goals

Type variables: <none>

\[
\begin{align*}
y_1: \text{int} \\
y_2: \text{int} \\
&m_1: \text{memory} \\
&m_2: \text{memory} \\
x_{\text{eq}}: M.x(m_1) = M.x(m_2) \\
y_{\text{eq}}: y_1 = y_2
\end{align*}
\]


\[\text{pre} = y(2) = y_2 \land M.x(2) = M.x(m_2) \land y(1) = y_1 \land M.x(1) = M.x(m_1)\]

\[M.f = M.f\]

\[\text{post} = \{\text{res}\}\]

and

Type variables: <none>

\[y_1: \text{int}\]
\[y_2: \text{int}\]
\[m_1: \text{memory}\]
\[m_2: \text{memory}\]
\[x_{eq}: M.x(m_1) = M.x(m_2)\]
\[y_{eq}: y_1 = y_2\]

\[y_2 = y_2 \land M.x(m_2) = M.x(m_2) \land y_1 = y_1 \land M.x(m_1) = M.x(m_1)\]

and

Type variables: <none>

\[y_1: \text{int}\]
\[y_2: \text{int}\]
\[m_1: \text{memory}\]
\[m_2: \text{memory}\]
\[x_{eq}: M.x(m_1) = M.x(m_2)\]
\[y_{eq}: y_1 = y_2\]

\[\forall \&1 \&2, \{\text{res}\} \Rightarrow \{\text{res}\} = 0 \iff \{\text{res}\} = 0\]

**Syntax:** \(\text{byequiv (}_ : P \Rightarrow Q)\). If the goal’s conclusion has the form

\[\text{Pr}[M_1.p_1(a_1_1,\ldots,a_{1,n_1}) \& m_1 : E_1] \Leftarrow \text{Pr}[M_2.p_2(a_2_1,\ldots,a_{2,n_2}) \& m_2 : E_2],\]

then \(\text{byequiv}\) behaves the same as in the first variant except that the conclusion of the third subgoal says that \(Q\) implies \(E_1(1) \Rightarrow E_2(2)\).

For example, if the current goal is

Type variables: <none>

\[\&m: \text{memory}\]

\[\text{Pr}[M.f() \& \&m : \text{res}] \Leftarrow \text{Pr}[N.f() \& \&m : \text{res}]\]

then running

\(\text{byequiv (}_ : \text{true} \Rightarrow \text{res}(1) \Rightarrow \text{res}(2)).\)

produces the goals

Type variables: <none>

\[\&m: \text{memory}\]

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

\[M.f = N.f\]

\[\text{post} = \{\text{res}\} \Rightarrow \{\text{res}\}\]
and

Type variables: <none>

&\text{m}: \text{memory}

true

and

Type variables: <none>

&\text{m}: \text{memory}

\forall k_1,k_2, (\text{res}(1) \Rightarrow \text{res}(2)) \Rightarrow \text{res}(1) \Rightarrow \text{res}(2)

And, if the current goal is

Type variables: <none>

&\text{m}: \text{memory}

pr[M.f() @ &\text{m} : \text{res}] \leq pr[N.f() @ &\text{m} : \text{res}]

then running

\text{byequiv}.

produces the goals

Type variables: <none>

&\text{m}: \text{memory}

\text{pre} = \text{true}

M.f \sim N.f

\text{post} = \text{res}(1) \Rightarrow \text{res}(2)

and

Type variables: <none>

&\text{m}: \text{memory}

\text{true}

and

Type variables: <none>

&\text{m}: \text{memory}

\forall k_1,k_2, (\text{res}(1) \Rightarrow \text{res}(2)) \Rightarrow \text{res}(1) \Rightarrow \text{res}(2)

**Syntax:** \text{byequiv} (\_ : P \Rightarrow Q). If the goal’s conclusion has the form

pr[M_1.p_1(a_{1,1},\ldots,a_{1,n_1}) @ k_{m_1} : E_1] \leq
pr[M_2.p_2(a_{2,1},\ldots,a_{2,n_2}) @ k_{m_2} : E_2] + pr[M_2.p_2(a_{2,1},\ldots,a_{2,n_2}) @ k_{m_2} : B_2],

then \text{byequiv} behaves the same as in the first variant except that the conclusion of the third subgoal says that \text{Q} implies !B_2(2) \Rightarrow E_1(1) \Rightarrow E_2(2).

For example, if the current goal is
CHAPTER 3. TACTICS

Type variables: <none>

@m: memory

\Pr[M.f() @ &m : res] <= \Pr[N.f() @ &m : res] + \Pr[N.f() @ &m : N.bad]

then running

\texttt{byequiv 
(_ : true ==⇒ ! N.bad{2} => res{1} => res{2}).}

produces the goals

Type variables: <none>

@m: memory

pre = true

M.f = N.f

post = !N.bad{2} => res{1} => res{2}

and

Type variables: <none>

@m: memory

ture

\textbf{FiXme Fatal: Why is the second subgoal pruned?} (Compare with first and second variants, where the corresponding subgoal isn’t pruned.) And, if the current goal is

Type variables: <none>

@m: memory

\Pr[M.f() @ &m : res] <= \Pr[N.f() @ &m : res] + \Pr[N.f() @ &m : N.bad]

then running

\texttt{byequiv.}

produces the goals

Type variables: <none>

@m: memory

pre = true

M.f = N.f

post = !N.bad{2} => res{1} => res{2}

and

Type variables: <none>

@m: memory

ture
**FiXme Fatal: Why is the second subgoal pruned?**

**Syntax:** `byequiv (_ : P =>> Q) : B1`. If the goal’s conclusion has the form

```plaintext
| Pr[M1.p1(a1,1,...,a1,n1) & &m1 : E1] - Pr[M2.p2(a2,1,...,a2,n2) & &m2 : E2] | <=
| Pr[M2.p2(a2,1,...,a2,n2) & &m2 : B2],
```

then `byequiv` behaves the same as in the first variant except that the conclusion of the third subgoal says that \( Q \) implies

\[
(B_1{\{1\}} <=> B_2{\{2\}}) \land ! B_2{\{2\}} => (E_1{\{1\}} <=> E_2{\{2\}})
\]

For example, if the current goal is

```
Type variables: <none>

&n: memory

```

then running

```
byequiv (_ : true =>> M.bad{\{1\}} = N.bad{\{2\}} /\ (! N.bad{\{2\}} => ={\{res\}})) : M.bad
```

produces the goals

```
Type variables: <none>

&n: memory

pre = true
M.f - N.f
post = M.bad{\{1\}} = N.bad{\{2\}} /\ (!N.bad{\{2\}} => ={\{res\}})
```

and

```
Type variables: <none>

&n: memory

true
```

and

```
Type variables: <none>

&n: memory

forall &1 &2,
M.bad{\{1\}} = N.bad{\{2\}} /\ (!N.bad{\{2\}} => ={\{res\}}) =>
(M.bad{\{1\}} <=> N.bad{\{2\}}) /\ (!N.bad{\{2\}} => res{\{1\}} <=> res{\{2\}})
```

Given the lemma

```
lemma L2 &n :
```

if the current goal is

```
Type variables: <none>

&n: memory

```
then running

\begin{verbatim}
byequiv M_N_f : M.bad.
\end{verbatim}

produces the goals

Type variables: <none>

\begin{verbatim}
&\text{m}: \text{memory}
true
\end{verbatim}

and

Type variables: <none>

\begin{verbatim}
forall &1 &2,
M\text{.}bad{1} = N\text{.}bad{2} \land (!N\text{.}bad{2} \Rightarrow \text{res}) \Rightarrow
(M\text{.}bad{1} \Leftrightarrow N\text{.}bad{2}) \land (!N\text{.}bad{2} \Rightarrow \text{res}{1} \Leftrightarrow \text{res}{2})
\end{verbatim}

\text{byphoare}

**Syntax:** \texttt{byphoare} \(_ : P \Rightarrow Q\). If the goal’s conclusion has the form

\[\text{Pr}[M.p(a_1, \ldots, a_n) \& \text{\&m : E}] = e,\]

reduce the goal to three subgoals:

- One with conclusion \texttt{phoare}[M.p : P \Rightarrow Q] = e;

- One whose conclusion says that P holds, where variables references are looked-up in \&m and references to the formal parameters of M.p have been replaced by its arguments; and

- One whose conclusion says that Q \Leftrightarrow E.

The argument to \texttt{byphoare} may be replaced by a proof term for \texttt{phoare}[M.p : P \Rightarrow Q] = e, in which case the first subgoal isn’t generated. Furthermore, either or both of P and Q may be replaced by \_, asking that the pre- or postcondition be inferred. Supplying no argument to \texttt{byphoare} is the same as replacing both P and Q by \_.

The other variants of the tactic behave similarly with regards to the use of proof terms and specification inference.

For example, consider the module

\begin{verbatim}
module M = {
  var x : int
  proc f(y : int) : int = {
    var z : int;
    x <= [x .. y];
    return x;
  }
}
\end{verbatim}

If the current goal is

Type variables: <none>

\begin{verbatim}
x' : int
y' : int
&\text{m}: \text{memory}
range: x' + 1 = y'
x'_\text{eq}: M.x{\text{m}} = x'
\end{verbatim}

\[\text{Pr}[M.f(y') \& \text{\&m : res} = x'] = 1/2r \lor 2/2r\]
then running

\texttt{byphoare \(_:\ M.x + 1 = y \land x' = M.x \Rightarrow x' = \text{res}).}

produces the goals

Type variables: <none>

\begin{align*}
\text{x': int} \\
\text{y': int} \\
\text{\&_m: memory} \\
\text{range: x' + 1 = y'} \\
\text{x_eq: M.x{m} = x'} \\
\text{pre = M.x + 1 = y \land x' = M.x} \\
M.f \\
\text{[=} 1%r / 2%r} \\
\text{post = x' = res}
\end{align*}

and

Type variables: <none>

\begin{align*}
\text{x': int} \\
\text{y': int} \\
\text{\&_m: memory} \\
\text{range: x' + 1 = y'} \\
\text{x_eq: M.x{m} = x'} \\
\text{M.x{m} + 1 = y' \land x' = M.x{m}} \\
\text{forall \&_hr, x' = \text{res}{\_hr} <=> \text{res}{\_hr} = x'}
\end{align*}

and

Type variables: <none>

\begin{align*}
\text{x': int} \\
\text{y': int} \\
\text{\&_m: memory} \\
\text{range: x' + 1 = y'} \\
\text{x_eq: M.x{m} = x'} \\
\text{forall \&_hr, x' = \text{res}{\_hr} <=> \text{res}{\_hr} = x'}
\end{align*}

Given the lemma

\texttt{lemma M\_f (x': int) :} \\
\texttt{phoare[M.f : M.x + 1 = y \land x' = M.x \Rightarrow x' = \text{res}] = (1%r / 2%r).}

if the current goal is

Type variables: <none>

\begin{align*}
\text{x': int} \\
\text{y': int} \\
\text{\&_m: memory} \\
\text{range: x' + 1 = y'} \\
\text{x_eq: M.x{m} = x'} \\
\text{Pr[M.f(y')] @ \&_m : \text{res} = x']} = 1%r / 2%r
\end{align*}

then running
byphoare (M.f x').

produces the goals

Type variables: <none>

x': int
y': int
&m: memory
range: x' + 1 = y'
x_eq: M.x{m} = x'

M.x{m} + 1 = y' \land x' = M.x{m}

and

Type variables: <none>

x': int
y': int
&m: memory
range: x' + 1 = y'
x_eq: M.x{m} = x'

forall &hr, x' = res{hr} <=> res{hr} = x'

Syntax: byphoare (_, P ==> Q). If the goal’s conclusion has the form

e <= Pr[M.p(a1,...,an) @ &m : E],

then byphoare behaves as in the first variant except the conclusion of the first subgoal is

phoare[M.p : P ==> Q] >= e.

For example, if the current goal is

Type variables: <none>

&m: memory

1%r / 2%r <= Pr[M.f() @ &m : res]

then running

byphoare (_, true ==> res).

produces the goals

Type variables: <none>

&m: memory

pre = true

M.f
[>=] 1%r / 2%r

post = res

and

Type variables: <none>
and

Type variables: <none>

\( \forall m, \text{true} \)

**FiXme** Fatal: It's confusing how the third goal has been simplified, but not pruned.

Given the lemma

\[
\text{lemma } M_f \ : \ \text{phoare}[M.f \ : \ \text{true} \implies \text{res}] \geq (1/2) / 2 \frac{r}{p}.
\]

if the current goal is

Type variables: <none>

\( \frac{1}{2}r / 2\frac{r}{p} \leq \text{Pr}[M.f() @ m : \text{res}] \)

then running

\text{byphoare } M_f.

produces the goals

Type variables: <none>

\( \forall m, \text{true} \)

And, if the current goal is

Type variables: <none>

\( \frac{1}{2}r / 2\frac{r}{p} \leq \text{Pr}[M.f() @ m : \text{res}] \)

then running

\text{byphoare.}

produces the goals

Type variables: <none>

\( \text{pre} = \text{true} \)
M.f
[>=] 1%r / 2%r

post = \texttt{res}

and

Type variables: <none>

\&m: memory

true

and

Type variables: <none>

\&m: memory

\texttt{forall }_, \texttt{true}

Syntax: \texttt{byphoare }(_ : P \implies Q). If the goal’s conclusion has the form

Pr[M.p(a_1,\ldots,a_n) \& \&m : E] <= e,

then \texttt{byphoare } behaves as in the first variant except the conclusion of the first subgoal is

phoare[M.p : P \implies Q] <= e.

For example, if the current goal is

Type variables: <none>

\&m: memory

Pr[M.f() \& \&m : \texttt{res}] <= 1\%r / 2\%r

then running

\texttt{byphoare }(_ : \texttt{true} \implies \texttt{res}).

produces the goals

Type variables: <none>

\&m: memory

\texttt{pre} = \texttt{true}

M.f
[<=] 1\%r / 2\%r

post = \texttt{res}

and

Type variables: <none>

\&m: memory

\texttt{true}

and
**Fixme** Fatal: It’s confusing how the third goal has been simplified, but not pruned.

Given the lemma

\[
\text{lemma } M_f : \text{phoare}[M.f : \text{true} \Rightarrow \text{res}] \leq (1/r / 2/r).
\]

if the current goal is

Type variables: <none>

\[\forall m : \text{memory} \]

\[\text{Pr}[M.f() \odot m : \text{res}] \leq 1/r / 2/r\]

then running

\texttt{byphoare } M_f.

produces the goals

Type variables: <none>

\[\forall m : \text{memory}\]

\[\text{true}\]

and

Type variables: <none>

\[\forall m : \text{memory}\]

\[\forall _, \text{true}\]

And, if the current goal is

Type variables: <none>

\[\forall m : \text{memory}\]

\[\text{Pr}[M.f() \odot m : \text{res}] \leq 1/r / 2/r\]

then running

\texttt{byphoare}.

produces the goals

Type variables: <none>

\[\forall m : \text{memory}\]

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

\[M.f \quad \leq 1/r / 2/r\]

\[\text{post} = \text{res}\]
and

\begin{verbatim}
Type variables: <none>
&m: memory
true
\end{verbatim}

and

\begin{verbatim}
Type variables: <none>
&m: memory
forall _, true
\end{verbatim}

\(\textcircled{\textsc{bypr}}\)

**Syntax:** \textsc{bypr} \(e_1 e_2\). If the goal’s conclusion has the form

\[
equiv[M.p - N.q : P \implies Q],
\]

and the \(e_i\) are expressions of the same type possibly involving memories \(\&1\) and \(\&2\) for \(M.p\) and \(N.q\), respectively, then reduce the goal to two subgoals:

- One whose conclusion says that for all memories \(\&1\) and \(\&2\) for \(M.p\) and \(N.q\), if \(e_1 = e_2\), then \(Q\) holds; and

- One whose conclusion says that, for all memories \(\&1\) and \(\&2\) for \(M.p\) and \(N.q\) and values \(a\) of the common type of the \(e_i\), if \(P\) holds, then the probability of running \(M.p\) in memory \(\&1\) and with arguments consisting of the values of its formal parameters in \(\&1\) and terminating in a memory in which the value of \(e_1\) (replacing references to \(\&1\) with reference to this memory) is \(a\) is the same as the probability of running \(N.q\) in memory \(\&2\) and with arguments consisting of the values of its formal parameters in \(\&2\) and terminating in a memory in which the value of \(e_2\) (replacing references to \(\&2\) with reference to this memory) is \(a\).

For example, consider the modules

\begin{verbatim}
module M = {
  var x : bool
  proc f(y : bool) : bool = {
    var b : bool;
    b <$ {0,1};
    return b /\ y;
  }
}.

module N = {
  var x : bool
  proc f(y : bool) : bool = {
    var b, b' : bool;
    b <$ {0,1};
    b' <$> {0,1};
    return (b ^^ b') /\ y;
  }
}.
\end{verbatim}

If the current goal is

\begin{verbatim}
Type variables: <none>
\end{verbatim}
CHAPTER 3. TACTICS

pre = \( \text{=} \{y\} \land M.x\{1\} = N.x\{2\}\)

\( M.f = N.f \)

post = \( \text{(res}\{1\} \land M.x\{1\}) = (\text{res}\{2\} \land N.x\{2\}) \)

then running

bypr (res, M.x)\{1\} (res, N.x)\{2\}.

produces the goals

Type variables: <none>

forall &1 &2 (a : bool * bool),
\( (\text{res}\{1\}, M.x\{1\}) = a \Rightarrow (\text{res}\{2\}, N.x\{2\}) = a \Rightarrow (\text{res}\{1\} \land M.x\{1\}) = (\text{res}\{2\} \land N.x\{2\}) \)

and

Type variables: <none>

forall &1 &2 (a : bool * bool),
\( =\{y\} \land M.x\{1\} = N.x\{2\} \Rightarrow \text{Pr}[M.f(y) \land &1 : (\text{res}, M.x) = a] = \text{Pr}[N.f(y) \land &2 : (\text{res}, N.x) = a] \)

Syntax: bypr. If the goal’s conclusion has the form

\( \text{hoare}[M.p : P \Rightarrow Q] \),

then reduce the goal to one whose conclusion says that, for all memories \( km \) for \( M.p \) such that \( P(m) \) holds, the probability of running \( M.p \) in memory \( km \) and with arguments consisting of the values of its formal parameters in \( km \) and terminating in a memory satisfying \( !Q \) is 0.

For example, consider the module

module M = {
  var x : bool
  proc f(y : bool) : bool = {
    var z : bool;
    z <$> \{0,1\};
    return x \land y \land z;
  }
}.

If the current goal is

Type variables: <none>

pre = !y \lor !M.x

\( M.f \)

post = !res

then running

bypr.
produces the goal

\[
\text{forall } \&m, !y(m) \lor !M.x(m) \Rightarrow Pr[M.f(y(m)) @ \&m : ! \text{res}] = 0\%
\]

\section*{exists*}

\textbf{Syntax:} \texttt{exists*} \(e_1, \ldots, e_n\). If the goal's conclusion is a \texttt{pRHL}, \texttt{HL} or \texttt{pHL} judgment or statement judgements and the \(e_i\) are well-typed expressions typically involving program variables (in the \texttt{pRHL} case, the expressions will refer to memories \&1 and \&2), then change the conclusion's precondition \(P\) to

\[
\text{exists} \ (x_1 \ldots x_n), x_1 = e_2 \lor \ldots \lor x_n = e_n \lor P.
\]

The tactic can be used in conjunction with \texttt{elim*} (p. 166) when handling a procedure call using a lemma that refers to initial values of program variables. See \texttt{elim*} (p. 166) for an example of this.

For example, if the current goal is

\begin{verbatim}
Type variables: <none>

pre = ={y} \land M.x{1} = N.x{2}  
M.f \neq N.f

post = ={res}
\end{verbatim}

then running

\begin{verbatim}
\texttt{exists*} \ (y + 1){1}, \ (N.x{2} - y{1})
\end{verbatim}

produces the goal

\begin{verbatim}
Type variables: <none>

pre =  
\texttt{exists} \ (f f0 : int),  
\ (f = y{1} + 1 \lor f0 = N.x{2} - y{1}) \lor 
={y} \lor M.x{1} = N.x{2}  
M.f \neq N.f

post = ={res}
\end{verbatim}

If the current goal is

\begin{verbatim}
Type variables: <none>

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

pre = ={y} \land M.x{1} = N.x{2}  
z \gets -(M.x) + y  \ (1) \ z \gets N.x - y

post = M.x{1} + z{1} + M.x{1} = y{2} + z{2} + y{2}
\end{verbatim}
then running

\[ \text{exists} \ast (y + 1) \{1\}, (N.x\{2\} - y\{1\}). \]

produces the goal

\[
\begin{align*}
& \text{Type variables: <none>} \\
& \&1 \text{ (left) } : M.f \\
& \&2 \text{ (right) } : N.f \\
& \pre = \text{exists } (f f0 : \text{int}), \\
& \quad (f = y\{1\} + 1 \land f0 = N.x\{2\} - y\{1\}) \land \\
& \quad \{y\} \land M.x\{1\} = N.x\{2\} \\
& z \leftarrow (-M.x) + y \quad \text{(1) } z \leftarrow N.x - y \\
& \post = M.x\{1\} + z\{1\} + M.x\{1\} = y\{2\} + z\{2\} + y\{2\}
\end{align*}
\]

If the current goal is

\[
\begin{align*}
& \text{Type variables: <none>} \\
& \pre = 0 \leq y \land 0 \leq M.x \\
& \quad M.f \\
& \post = 0 \leq \text{res}
\end{align*}
\]

then running

\[ \text{exists} \ast (y + M.x - 2). \]

produces the goal

\[
\begin{align*}
& \text{Type variables: <none>} \\
& \pre = \text{exists } (f : \text{int}), f = y + M.x - 2 \land 0 \leq y \land 0 \leq M.x \\
& \quad M.f \\
& \post = 0 \leq \text{res}
\end{align*}
\]

If the current goal is

\[
\begin{align*}
& \text{Type variables: <none>} \\
& \text{Context : } M.f \\
& \pre = 0 \leq y \land 0 \leq M.x \\
& \quad (1) z \leftarrow (-M.x) + y \\
& \post = 0 \leq M.x + z + M.x
\end{align*}
\]

then running

\[ \text{exists} \ast (y + M.x - 2). \]
produces the goal

```
Type variables: <none>

Context : M.f

pre = exists (f : int), f = y + M.x - 2 \ 0 <= y \ 0 <= M.x

(1)  \ z \leftarrow (-M.x) + y

post = 0 <= M.x + z + M.x
```

If the goal’s conclusion is a pRHL, HL or pHL judgement or statement judgement whose precondition has the form

```
exists (x_1 \ldots x_n), P,
```

then remove the existential quantification from the precondition, and universally quantify the judgement or statement judgement by the \( x_i \).

Such existential quantifications may be introduced by \texttt{sp} (p. 81) or \texttt{exists} (p. 164).

For example, if the current goal is

```
Type variables: <none>

pre =
exists (f f0 : int),
(f = y{1} + 1 /\ f0 = N.x{2} - y{1}) /\
={y} /\ M.x{1} = N.x{2}
M.f ~ N.f

post = ={res}
```

then running

```
elim*.
```
produces the goal

```
Type variables: <none>

forall (f f0 : int),
equiv[ M.f = N.f :
(f = y{1} + 1 /\ f0 = N.x{2} - y{1}) /\
 =\{y\} /\ M.x{1} = N.x{2} ==\{res\}]
```

If the current goal is

```
Type variables: <none>

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

pre =
exists (f f0 : int),
(f = y{1} + 1 /\ f0 = N.x{2} - y{1}) /\
```
={y} \land M.x(1) = N.x(2)

z \leftarrow (-M.x) + y 

(1) z \leftarrow N.x - y

post = M.x(1) + z(1) + M.x(1) = y(2) + z(2) + y(2)

then running

elim*.

produces the goal

Type variables: <none>

forall (f f0 : int),

equiv{ M.f.z \leftarrow (-M.x(hr)) + M.f.y(hr) - N.f.z <-

N.x(hr) - N.f.y(hr) :

(f = y(1) + 1 \land f0 = N.x(2) - y(1)) \land

={y} \land M.x(1) = N.x(2) ==>

M.x(1) + z(1) + M.x(1) = y(2) + z(2) + y(2)}

If the current goal is

Type variables: <none>

pre = exists (f : int), f = y + M.x - 2 /\ 0 \leq y /\ 0 \leq M.x

M.f

post = 0 \leq res

then running

elim*.

produces the goal

Type variables: <none>

forall (f : int),

hoare[ M.f : f = y + M.x - 2 /\ 0 \leq y /\ 0 \leq M.x ==>. 0 \leq res]

If the current goal is

Type variables: <none>

Context : M.f

pre = exists (f : int), f = y + M.x - 2 /\ 0 \leq y /\ 0 \leq M.x

(1) z \leftarrow (-M.x) + y

post = 0 \leq M.x + z + M.x

then running

elim*.

produces the goal
As a more realistic example, consider the module

```
module M = {
    proc f(x : int) : int = {
        return x + 1;
    }
    proc g(x : int) : int = {
        var y : int;
        y <@ f(x);
        return y;
    }
}.
```

and lemma

```
lemma M_f (x : int) : hoare[M.f : x = x => res = x + 1].
```

If the current goal is

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

```
Context : M.g
pre = x = -1
(1) y <@ M.f(x)
post = y = 0
```

then running

```
exists* x.
```

produces the goal

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

```
Context : M.g
pre = exists (x0 : int), x0 = x /\ x = -1
(1) y <@ M.f(x)
post = y = 0
```

from which running

```
elim=> x'.
```

produces the goal

```
Type variables: <none>
```
x’: int

Context : M.g
pre = x’ = x /\ x = -1

(1) y <@ M.f(x)
post = y = 0

from which running

\textbf{call} \langle M.f \; x’ \rangle.

produces the goal

Type variables: <none>
x’: int

Context : M.g
pre = x’ = x /\ x = -1

post =

x’ = x &k forall (result : int), result = x’ + 1 => result = 0

\textcircled{○} \textbf{hoare}

\textbf{FiXme Fatal: Update waiting for overhaul of pHL.}

\textbf{Syntax:} \texttt{hoare <spec>}. Derives a null probability from a HL judgement on the procedure involved. \texttt{hoare} can also be used to derive pHL judgments and certain probability inequalities by automatically applying \texttt{conseq} (p. 136).

\textbf{Examples:}

\begin{align*}
\texttt{hoare} & \quad \texttt{hoare} \\
\{\text{true}\} & \quad \{\text{true}\} \\
\mathcal{P} & \quad \mathcal{P} \\
\{\text{f} \{\neg Q\}\} & \quad \{\text{f} \{\neg Q\}\} \\
\mathcal{P}[m, \mathcal{f}(\vec{x}) : Q] = 0 & \quad \mathcal{P}[m, \mathcal{f}(\vec{x}) : Q] = 0 \\
\end{align*}

\textcircled{○} \textbf{phoare split}

\textbf{FiXme Fatal: Update waiting for overhaul of pHL.}

\textbf{Syntax:} \texttt{phoare split} \texttt{\delta_A} \texttt{\delta_B} \texttt{\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*}
\texttt{phoare split} & \quad \texttt{phoare split} \\
\texttt{\delta_A} & \quad \texttt{\delta_A} \\
\texttt{\delta_B} & \quad \texttt{\delta_B} \\
\texttt{\delta_{AB}} & \quad \texttt{\delta_{AB}} \\
\end{align*}
Syntax: **phoare split** ![Formula]

Splits a pHL judgment into two judgments whose postcondition are true and the negation of the original postcondition, respectively.

**Examples:**

\[
\begin{align*}
\text{phoare split } & \delta_1 \delta_2 \\
\rightarrow & \delta_1 \delta_2
\end{align*}
\]

\[
\begin{align*}
\{P\} \{\text{true}\} & \circ \delta_1 \\
\{P\} \{\neg Q \circ^{-1} \delta_1\} & \circ \delta_2
\end{align*}
\]

Syntax: **phoare split** ![Formula]

Splits a pHL judgment following an event \( A \).

**Examples:**

\[
\begin{align*}
\text{phoare split } & \delta_A \delta_{A'} \\
\rightarrow & \delta_A \delta_{A'}
\end{align*}
\]

\[
\begin{align*}
\{P\} \{Q \land A\} & \circ \delta_A \\
\{P\} \{Q \land \neg A\} & \circ \delta_{A'}
\end{align*}
\]

3.4.4 Automated Tactics

**exfalso.** Combines **conseq** (p. 136), **byequiv** (p. 149), **byphoare** (p. 156), **hoare** (p. 169) and **bypr** (p. 162) to strengthen the precondition into false and discharge the resulting trivial goal.

For example, if the current goal is

Type variables: <none>

\[
\begin{align*}
&\text{seq 3 3 : (=i} \
&0 < i{1} \
i{1} <= 0).
\end{align*}
\]

then running

produces the goals

Type variables: <none>

\[
\begin{align*}
&\text{seq 3 3 : (=i} \
&0 < i{1} \
i{1} <= 0).
\end{align*}
\]
pre = true

\[\begin{align*}
  i &\leftarrow 0 \\
  j &\leftarrow 0 \\
  \textbf{while} (i \leq 0) \{ \\
    i &\leftarrow i - 1 \\
  \} \\
\end{align*}\]

\[\begin{align*}
  j &\leftarrow 0 \\
  \textbf{while} (i \leq 0) \{ \\
    i &\leftarrow i - 1 \\
  \} \\
\end{align*}\]

\[\begin{align*}
  \text{post} = &\{i\} /\ \& 0 < i(1) /\ \& i(1) \leq 0
\end{align*}\]

and

Type variables: <none>

\[\begin{align*}
  k1 &\text{ (left)} : M.f \\
  k2 &\text{ (right)} : N.f
\end{align*}\]

\[\begin{align*}
  \text{pre} = &\{i\} /\ \& 0 < i(1) /\ \& i(1) \leq 0 \\
  \textbf{while} (j < 10) \{ \\
    i &\leftarrow i + 2 \\
    j &\leftarrow j + 1 \\
  \} \\
\end{align*}\]

\[\begin{align*}
  \text{post} = &\{i\}
\end{align*}\]

The first of these goals is solved by running

\[\begin{align*}
  \text{while} (\neg i) /\ \& i(1) \leq 0); \text{ auto; smt.}
\end{align*}\]

And running

\[\text{exfalso.}\]

reduces the second of these goals to

Type variables: <none>

\[\begin{align*}
  \forall k1, k2, &\{i\} /\ \& 0 < i(1) /\ \& i(1) \leq 0 \Rightarrow \text{false}
\end{align*}\]

which \textit{smt} solves.

\textbf{Fixme Fatal: Perhaps need other examples?}

\[\text{auto}\]

If the current goal is a PRHL, HL or PHL statement judgement, uses various program logic tactics in an attempt to reduce the goal to a simpler one. Never fails, but may fail to make any progress.

\textbf{Fixme Fatal: Need better description of when the tactic is applicable and how the tactic works!}

For example, if the current goal is

Type variables: <none>

\[\begin{align*}
  \text{Context} &:\ M.f \\
  \text{pre} &:= 1 \leq y < x
\end{align*}\]
(1--) if \((y < x)\) {  
(1.1) \(z_1 \leftarrow x\)  
(1.2) \(x \leftarrow y\)  
(1.3) \(y \leftarrow z_1\)  
(1--) }  
(2--) \(z_1 \llbracket x .. y \rrbracket\)  
(3--) \(z_2 \llbracket x - 1 .. y + 1 \rrbracket\)

\[
\text{post} = 0 \leq z_1 + z_2
\]

then running

\textit{auto.}

produces the goal

\[
\begin{align*}
\text{Type variables: } & \langle \text{none} \rangle \\
& \forall \& hr, \quad 1 \leq y^{\langle hr \rangle} < x^{\langle hr \rangle} \Rightarrow \\
& \text{if } y^{\langle hr \rangle} < x^{\langle hr \rangle} \text{ then} \\
& \quad \forall (z_1 : \text{int}), \\
& \quad \; z_1 \in [y^{\langle hr \rangle} .. x^{\langle hr \rangle}] \Rightarrow \\
& \quad \quad \forall (z_2 : \text{int}), \\
& \quad \; \; z_2 \in [y^{\langle hr \rangle} - 1 .. x^{\langle hr \rangle} + 1] \Rightarrow 0 \leq z_1 + z_2 \\
& \quad \text{else} \\
& \quad \quad \forall (z_1 : \text{int}), \\
& \quad \; \; z_1 \in [x^{\langle hr \rangle} .. y^{\langle hr \rangle}] \Rightarrow \\
& \quad \quad \forall (z_2 : \text{int}), \\
& \quad \; \; \; z_2 \in [x^{\langle hr \rangle} - 1 .. y^{\langle hr \rangle} + 1] \Rightarrow 0 \leq z_1 + z_2
\end{align*}
\]

which \textit{progress; smt} is able to solve. If the current goal is

\[
\begin{align*}
\text{Type variables: } & \langle \text{none} \rangle \\
\text{Context : } & \text{N.f} \\
\text{pre} = & \; 1 \leq y < x \\
(1) & \quad z_1 \llbracket x .. y \rrbracket \\
(2) & \quad x \leftarrow x - 1 \\
(3) & \quad y \leftarrow y + 1 \\
(4) & \quad z_2 \llbracket x .. y \rrbracket \\
(5) & \quad x \leftarrow x - 1 \\
(6) & \quad y \leftarrow y + 1 \\
\text{post} = & \; 0 \leq z_1 + z_2
\end{align*}
\]

then running

\textit{auto.}

produce a single goal, which \textit{smt} is able to solve. And, if the current goal is

\[
\begin{align*}
\text{Type variables: } & \langle \text{none} \rangle \\
& \& 1 \text{ (left) } : \text{M.f} \\
& \& 2 \text{ (right) } : \text{N.f}
\end{align*}
\]
pre = \( (x, y) \land 1 < x \leq y \)

\[
\begin{align*}
\text{if } (y < x) & \{ \\
    z_1 & \leftarrow x & (1.1) \\
    x & \leftarrow y & (1.2) \\
    y & \leftarrow z_1 & (1.3) \\
\} \\
    z_1 & <\ [x..y] & (1--) \\
    z_2 & <\ [x-1..y+1] & (2--) \\
\end{align*}
\]

post = \( z_1 + z_2 \oidalop \)

then running

\texttt{auto}.

produce a single goal, which \texttt{smt} is able to solve.

\texttt{\# sim}

\texttt{sim} attempts to solve a goal whose conclusion is a PRHL judgement or statement judgement by working backwards, propagating and extending a conjunction of equalities between variables of the two programs, verifying that the conclusion’s precondition implies the final conjunction of equalities. It’s capable of working backwards through \texttt{if} and \texttt{while} statements and handling random assignments, but only when the programs are sufficiently similar (thus its name). Sometimes this process only partly succeeds, leaving a statement judgement whose programs are prefixes of the original programs.

\textbf{Syntax:} \texttt{sim}. Without any arguments, \texttt{sim} attempts to infer the conjunction of program variable equalities from the conclusion’s postcondition.

For example, if the current goal is

\begin{align*}
\text{Type variables: } & \texttt{<none>} \\
& \texttt{\#1 (left) : M.f} \\
& \texttt{\#2 (right) : N.f} \\
\pre & = x \land y + 1 = N.v \\
i & \leftarrow 0 & (1--) \\
y & \leftarrow y + 1 & (2--) \\
\text{while } (j < N.v) & \{ \\
    & N.u \leftarrow N.u + N.u & (2.1) \\
    & j \leftarrow j + 1 & (2.2) \\
\} & (2--) \\
\text{while } (i < y) & \{ \\
    & x \leftarrow x + x & (3.1) \\
    & i \leftarrow i + 1 & (3.2) \\
\} & (3--) \\
\text{post} & = x \oidop \]
\end{align*}

then running

\texttt{\# sim}.

produces the goal

\begin{align*}
\text{Type variables: } & \texttt{<none>} \\
\end{align*}
\&1 \text{(left)} : M.f \\
\&2 \text{(right)} : N.f

\[ \text{pre} = x(1) = N.u(2) \land y(1) + 1 = N.v(2) \]

\[ i \leftarrow 0 \quad (1) \quad j \leftarrow 0 \]

\[ y \leftarrow y + 1 \quad (2) \]

\[ \text{post} = x(1) = N.u(2) \land y(1) = N.v(2) \land i(1) = j(2) \]

which \textit{auto} is able to solve.

\textbf{Syntax:} \texttt{sim} / \phi : \texttt{eqs}. One may give the starting conjunction, \texttt{eqs}, of equalities explicitly, and may also specify an invariant \phi on the global variables of the programs.

For example, if the current goal is

\textbf{Type variables:} <none>

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

\[ \text{pre} = x(1) = N.u(2) \land y(1) + 1 = N.v(2) \land 0 \leq y(1) \]

\[ i \leftarrow 0 \quad (1--) \quad j \leftarrow 0 \]

\[ y \leftarrow y + 1 \quad (2--)
\]

\[ \text{while} \ (j < N.v) \{ \]
\[ \quad (2.1) \quad N.u \leftarrow N.u + N.u \]
\[ \quad (2.2) \quad j \leftarrow j + 1 \]
\[ \} \quad (2--) \]

\[ \text{while} \ (i < y) \{ \]
\[ \quad (3--) \quad x \leftarrow x + x \]
\[ \quad (3.1) \quad i \leftarrow i + 1 \]
\[ \} \quad (3--)
\]

\[ \text{post} = x(1) = N.u(2) \land 1 < N.v(2) + 1 \]

then running

\[ \texttt{sim} / (0 < N.v(2)) : (x(1) = N.u(2) \land y(1) = N.v(2)) . \]

produces the goals

\textbf{Type variables:} <none>

\[ \forall \&1 \&2, \]
\[ \quad (x(1) = N.u(2) \land y(1) = N.v(2)) \land 0 < N.v(2) \Rightarrow \]
\[ \quad x(1) = N.u(2) \land 1 < N.v(2) + 1 \]

\textbf{and}

\textbf{Type variables:} <none>

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

\[ \text{pre} = x(1) = N.u(2) \land y(1) + 1 = N.v(2) \land 0 \leq y(1) \]

\[ i \leftarrow 0 \quad (1) \quad j \leftarrow 0 \]
\[ y \gets y + 1 \quad (2) \]

\[
\text{post} = \left( x(1) = N.u(2) \land y(1) = N.v(2) \land i(1) = j(2) \right) \land 0 < N.v(2)
\]

which \texttt{smt} and \texttt{auto;smt}, respectively, are able to solve.

**Syntax:** \texttt{sim procseq\_1 \ldots procseq\_\_ / / eqs}. In its most general form, one may also supply a sequence of procedure global equality specifications of the form

\[(M.p - N.q : eqs),\]

where \texttt{eqs} is a conjunction of global variable equalities. When \texttt{sim} encounters a pair of procedure calls consisting of a call to \(M.p\) in the first program and \(N.q\) in the second program, it will generate a subgoal whose conclusion is a \texttt{pRHL} judgment between \(M.p\) and \(N.q\), whose precondition assumes equality of its arguments, \texttt{eqs} and \(\phi\), and whose postcondition requires equality of the calls’ results, \texttt{eqs} and \(\phi\).

One may also replace \(M.p - N.q\) by \texttt{_}, meaning that the same conjunction of global variable equalities is used for all procedure calls.

For example, if the current goal is

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

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

\[
\begin{align*}
\text{pre} & = \text{true} \\
& k <- 0 \quad (1--) \quad k <- 0 \\
& M.x <- 0 \quad (2--) \quad N.x <- 0 \\
& M.i <- 0 \quad (3--) \quad N.i <- 0 \\
& M.j <- 0 \quad (4--) \quad N.j <- 0 \\
\text{while } (k < 10) \{ \\
& \quad M.f() \quad (5.1) \quad N.f() \\
& \quad k <- k + 1 \quad (5.2) \quad k <- k + 1 \\
\} \quad (5--) \} \\
\text{while } (k < 20) \{ \\
& \quad M.g() \quad (6.1) \quad N.g() \\
& \quad k <- k + 1 \quad (6.2) \quad k <- k + 1 \\
\} \quad (6--) \}
\]

\[
\text{post} = \text{\textasciitilde}\{M.x(1) + k(1)\} \text{\textasciitilde}\{N.x(2) - k(2)\}
\]

then running

\[
\texttt{sim} (M.f - N.f : M.i(1) = N.i(2)) \\
\quad (M.g - N.g : M.j(1) = N.j(2)) / \\
\quad (N.x(1) = -N.x(2)) : (=k) .
\]

produces the goals

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

\[
\texttt{forall } &1 \ &2, \\
\quad =\{k\} \land M.x(1) = -N.x(2) \Rightarrow \text{\textasciitilde}\{M.x(1) + k(1)\} \text{\textasciitilde}\{N.x(2) - k(2)\}
\]

and

\[
\text{Type variables: <none>}
\]
pre = true /\ M.i{1} = N.i{2} /\ M.x{1} = -N.x{2}

M.f ~ N.f

post = =\{res\} /\ M.i{1} = N.i{2} /\ M.x{1} = -N.x{2}

and

Type variables: <none>

pre = true /\ M.j{1} = N.j{2} /\ M.x{1} = -N.x{2}

M.g ~ N.g

post = =\{res\} /\ M.j{1} = N.j{2} /\ M.x{1} = -N.x{2}

and

Type variables: <none>

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

pre = true

k <= 0  \hspace{1cm} (1) k <= 0
M.x <= 0 \hspace{1cm} (2) N.x <= 0

post = =\{k\} /\ M.x{1} = -N.x{2}

which smt, proc;auto;smt, proc;auto;smt and auto, respectively, are able to solve.

And, if the current goal is

pre = true

k <= 0  \hspace{1cm} (1--) k <= 0
M.x <= 0 \hspace{1cm} (2--) N.x <= 0
M.i <= 0 \hspace{1cm} (3--) N.i <= 0
M.j <= 0 \hspace{1cm} (4--) N.j <= 0

while (k < 10) { \hspace{1cm} (5--) while (k < 10) {
M.f()  \hspace{1cm} (5.1) N.f()
\[ \hspace{1cm} \]
\[ \hspace{1cm} \] k <= k + 1 \hspace{1cm} (5.2) k <= k + 1
}\hspace{1cm} (5--) }

while (k < 20) { \hspace{1cm} (6--) while (k < 20) {
M.g()  \hspace{1cm} (6.1) N.g()
\[ \hspace{1cm} \]
\[ \hspace{1cm} \] k <= k + 1 \hspace{1cm} (6.2) k <= k + 1
}\hspace{1cm} (6--) }

post = `|M.x{1} + k{1}| = `|N.x{2} - k{2}|`

then running
produces the goals

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

\[
\text{forall } \&1, \&2,\]

\[
=\{k\} \land M.x(1) = -N.x(2) \Rightarrow \| M.x(1) + k(1) \| = \| N.x(2) - k(2) \|
\]

and

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

\[
\text{pre } = \]

\[
\text{true } \land (M.j(1) = N.j(2) \land M.i(1) = N.i(2)) \land M.x(1) = -N.x(2)
\]

\[
M.f - N.f
\]

\[
\text{post } = \]

\[
=\{\text{res}\} \land
\]

\[
(M.j(1) = N.j(2) \land M.i(1) = N.i(2)) \land M.x(1) = -N.x(2)
\]

and

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

\[
\text{pre } = \]

\[
\text{true } \land (M.j(1) = N.j(2) \land M.i(1) = N.i(2)) \land M.x(1) = -N.x(2)
\]

\[
M.g - N.g
\]

\[
\text{post } = \]

\[
=\{\text{res}\} \land
\]

\[
(M.j(1) = N.j(2) \land M.i(1) = N.i(2)) \land M.x(1) = -N.x(2)
\]

and

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

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

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

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

\[
(1) \ k \leftarrow 0
\]

\[
M.x \leftarrow 0
\]

\[
(2) \ N.x \leftarrow 0
\]

\[
\text{post } = \{k\} \land M.x(1) = -N.x(2)
\]

which \texttt{smt, proc:auto;sm}t, \texttt{proc:auto;sm}t and \texttt{auto}, respectively, are able to solve.

### 3.4.5 Advanced Tactic

\texttt{fel}

**Syntax:** \texttt{fel init ctr stepub bound bad conds inv} “fel” stands for “failure event lemma”. To use this tactic, one must load the theory \texttt{FelTactic}. To be applicable, the current goal’s conclusion must have the form
Here:

- $ub$ ("upper bound") is an expression of type `real`.
- $ctr$ is the counter, an expression of type `int` involving program variables.
- $bad$ is an expression of type `bool` involving program variables. It is the "bad" or "failure" event.
- $inv$ is an optional invariant on program variables; if it’s omitted, `true` is used.
- $init$ is a natural number no bigger than the number of statements in $M.p$. It is the length of the initial part of the procedure that “initializes” the failure event lemma—causing $ctr$ to become 0 and $bad$ to become `false` and establishing $inv$. The non-initialization part of the procedure may not directly use the program variables on which $ctr$, $bad$ and $inv$ depend. These variables may only be modified by concrete procedures $M.p$ may directly or indirectly call—such procedures are called oracle procedures. If $M.p$ directly or indirectly calls an abstract procedure, there must be a module constraint saying that the abstract procedure may not modify the program variables determining the values of $ctr$, $bad$ and $inv$ or that are used by the oracle procedures.
- $bound$ is an expression of type `int`. It must be the case that $\phi \land inv \Rightarrow bad \land ctr \leq bound$.
- $conds$ is a list of procedure preconditions
  
  $[N_1.p_1 : \phi_1; \ldots; N_l.p_l : \phi_l]$,

  where the $N_i.p_i$ are procedures, and the $\phi_i$ are expressions of type `bool` involving program variables and procedure parameters. When a procedure’s precondition is true, it must increase the counter’s value; when it isn’t true, it must not decrease the counter’s value, and must preserve the value of $bad$. Whether a procedure’s precondition holds or not, the invariant $inv$ must be preserved.
- $stepub$ is a function of type `int -> real`, providing an upper bound as a function of the counter’s current value. When a procedure’s precondition, the invariant $inv$ and $0 \leq ctr < bound$ hold, the probability that $bad$ becomes set during that call must be upper-bounded by the application of $stepub$ to the counter’s value. In addition, it must be the case that the summation of $stepub\ i$, as $i$ ranges from 0 to $bound - 1$, is upper-bounded by $ub$.

The subgoals generated by `fel` enforce the above rules. The best way to understand the details is via an example.

For example, consider the declarations

```plaintext
op upp : { int | 1 <= upp } as ge1_upp.
op n : { int | 0 <= n } as ge0_n.

module type OR = {
  proc init() : unit
  proc gen() : bool
  proc add(_ : int) : unit
}.

module Or : OR = {
  var won : bool
  var gens : int list
```

\[ \Pr[M.p(a_1, \ldots, a_r) @ km : \phi] \leq ub. \]
Here, the oracle has a boolean variable `won`, which is the bad event. It also has a list of integers `gens`, all of which are within the range 1 to `upp`, inclusive—the integers “generated” so far. The counter is the size of `gens`. The procedure `gen` randomly generates such an integer, setting `won` to `true` if the integer was previously generated. And the procedure `add` adds a new integer to the list of generated integers, without possibly setting `bad`. Both `gen` and `add` do nothing when the counter reaches the bound `n`. The adversary has access to both `gen` and `bad`.

If the current goal is

\[
\Pr[G(Adv).main() \& \&m : Or.won] <= (n \times (n - 1))\frac{r}{(2 \times upp)^r}
\]

then running

\[
\text{fel}
\]

1
(size Or.gens)
(fun ctr => ctr%r * (1%r / upp%r))
n\nOr.won
[Or.gen : (size Or.gens < n);
Or.add : (size Or.gens < n / \ 1 <= x <= upp)]
produces the goals

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

\[ \text{Adv: ADV(Or)} \]
\[ \text{&m: memory} \]

\[ \text{forall } \text{&m}, \]
\[ \text{Or.won} \{ \text{&m} \} \]
\[ \Rightarrow \]
\[ \text{size Or.gens} \{ \text{&m} \} \leq n \Rightarrow \text{Or.won} \{ \text{&m} \} \]
\( \land \)
\[ \text{size Or.gens} \{ \text{&m} \} \leq n \]

and

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

\[ \text{Adv: ADV(Or)} \]
\[ \text{&m: memory} \]

\[ \text{forall } \text{&m0}, \]
\[ \text{Or.won} \{ \text{&m0} \} \]
\[ \Rightarrow \]
\[ \text{size Or.gens} \{ \text{&m0} \} \leq n \Rightarrow \text{Or.won} \{ \text{&m0} \} \]
\( \land \)
\[ \text{size Or.gens} \{ \text{&m0} \} \leq n \]

and

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

\[ \text{Adv: ADV(Or)} \]
\[ \text{&m: memory} \]

\[ \text{Context: } G(\text{Adv}).\text{main} \]
\[ \text{pre} = \text{tt} = \text{tt} \]
\( \land \)
\[ \text{Or.gens} = \text{Or.gens} \{ \text{m} \} \]
\( \land \)
\[ \text{Or.won} = \text{Or.won} \{ \text{m} \} \]

(1) \text{Or.init}()

\[ \text{post} = (\text{(!Or.won} \land \text{size Or.gens} = 0) \land \text{size Or.gens} \leq n) \]

and

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

\[ \text{Adv: ADV(Or)} \]
\[ \text{&m: memory} \]

\[ \text{pre} = \]
\[ (((\text{0} \leq \text{size Or.gens} \land \text{size Or.gens} < n) \land \text{!Or.won}) \land \]
\[ \text{size Or.gens} \leq n) \land \]
\[ \text{size Or.gens} < n \land \]
\[ 1 \leq x \leq \text{upp} \]
\[ \text{Or.add} \]
\[ [\leq] \text{size Or.gens} \times (x \times \text{upp}) \]

\[ \text{post} = \text{Or.won} \]

and

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

\[ \text{Adv: ADV(Or)} \]
\[ \text{&m: memory} \]

\[ \text{forall } (c : \text{int}), \]
CHAPTER 3. TACTICS

hoare[ Or.add :
((size Or.gens < n \ 1 <= x <= upp) \ c = size Or.gens) \/
size Or.gens <= n ==>
c < size Or.gens \ size Or.gens <= n]

and

Type variables: <none>
Adv: ADV{Or}
&m: memory

forall (b : bool) (c : int),
hoare[ Or.add :
(! (size Or.gens < n \ 1 <= x <= upp) \/
Or.won = b \ size Or.gens = c) \/
size Or.gens <= n ==>
(Or.won = b \ c <= size Or.gens) \ size Or.gens <= n]

and

Type variables: <none>
Adv: ADV{Or}
&m: memory

pre = (((0 <= size Or.gens \ size Or.gens < n) \ !Or.won) \/
size Or.gens <= n) \ size Or.gens < n
Or.gen
[<=] size Or.gens%r * (1%r / upp%r)

post = Or.won

and

Type variables: <none>
Adv: ADV{Or}
&m: memory

forall (c : int),
hoare[ Or.gen :
(size Or.gens < n \ c = size Or.gens) \/
size Or.gens <= n ==>
c < size Or.gens \ size Or.gens <= n]

and

Type variables: <none>
Adv: ADV{Or}
&m: memory

forall (b : bool) (c : int),
hoare[ Or.gen :
(! size Or.gens < n \ Or.won = b \ size Or.gens = c) \/
size Or.gens <= n ==>
(Or.won = b \ c <= size Or.gens) \ size Or.gens <= n]
**CHAPTER 3. TACTICS**

**eager** is a family of tactics for proving PRHL statement judgements of the form

\[ \text{equiv}[s_1 t_1 - t_2 s_2 : P \Rightarrow Q], \]

where the pre- and postconditions \( P \) and \( Q \) are conjunctions of equalities between program variables, and the statement sequences \( s_i \) only read and write global variables. Here \( s_1 \) is in the “eager” position, and its replacement, \( s_2 \), is in the “lazy” position. Some of the tactics work with eager judgements of the form

\[ \text{eager}[s_1, M.p - N.q, s_2 : P \Rightarrow Q], \]

where, again, the \( s_i \) only involve global variables, but where \( P \) and \( Q \) may talk about the parameters and results of \( M.p \) and \( N.q \) in the usual way.

The context of our examples is the following EASYCRYPT script involving variable incrementation oracles \( \text{Or1} \) and \( \text{Or2} \), which only differ in that \( \text{Or2} \) keeps a “transcript” of its operation.

```plaintext
module type OR = {
proc init() : unit
proc incr_x() : unit
proc incr_y() : unit
proc incr_xy() : unit
proc incr_yx() : unit
proc loop(n : int) : bool
}.

module Or1 : OR = {
  var x, y : int
  var b : bool
  proc init() : unit = { x <- 0; y <- 0; b <- true; }
  proc incr_x() : unit = { x <- x + 1; }
  proc incr_y() : unit = { y <- y + 1; }
  proc incr_xy() : unit = { incr_x(); incr_y(); }
  proc incr_yx() : unit = { incr_y(); incr_x(); }
  proc loop(n : int) : bool = {
    while (0 < n) {
      if (b) incr_x(); else incr_y();
      b <- !b;
    }
    return !b;
  }
}.

module Or2 : OR = {
  var x, y : int
  var b : bool
  var trace : bool list
  proc init() : unit = { x <- 0; y <- 0; b <- true; trace <- []; }
  proc incr_x() : unit = { x <- x + 1; trace <- trace ++ [true]; }
  proc incr_y() : unit = { y <- y + 1; trace <- trace ++ [false]; }
  proc incr_xy() : unit = { incr_x(); incr_y(); }
  proc incr_yx() : unit = { incr_y(); incr_x(); }
  proc loop(n : int) : bool = {
    while (0 < n) {
      if (b) incr_x(); else incr_y();
      b <- !b;
    }
    return !b;
  }
}.
```
CHAPTER 3. TACTICS

\[\text{lemma} \text{ eager_incr :}
\text{  eager}[\text{Or1.incr}_x(); \text{Or1.incr}_y - \text{Or2.incr}_y, \text{Or2.incr}_x(); : 
  =\{x, y\}(\text{Or1, Or2}) \implies =\{x, y\}(\text{Or1, Or2}).
\]
\text{proof.}
\text{eager proc.}
\text{inline \*; auto.}
\text{qed.}

\[\text{lemma} \text{ eager_incr_x :}
\text{  eager}[\text{Or1.incr}_x(); \text{Or1.incr}_y();, \text{Or1.incr}_x -
  \text{Or2.incr}_x, \text{Or2.incr}_y(); \text{Or2.incr}_x(); : 
  =\{x, y, b\}(\text{Or1, Or2}) \implies =\{x, y, b\}(\text{Or1, Or2}).
\]
\text{proof.}
\text{proc\*).
\text{inline\*; auto.}
\text{qed.}

\[\text{lemma} \text{ eager_incr_y :}
\text{  eager}[\text{Or1.incr}_x(); \text{Or1.incr}_y();, \text{Or1.incr}_y -
  \text{Or2.incr}_y, \text{Or2.incr}_y(); \text{Or2.incr}_x(); : 
  =\{x, y, b\}(\text{Or1, Or2}) \implies =\{x, y, b\}(\text{Or1, Or2}).
\]
\text{proof.}
\text{proc\*; inline\*; auto.}
\text{qed.}

\[\text{lemma} \text{ eager_incr_xy :}
\text{  eager}[\text{Or1.incr}_x(); \text{Or1.incr}_y();, \text{Or1.incr}_xy -
  \text{Or2.incr}_xy, \text{Or2.incr}_y(); \text{Or2.incr}_x(); :
  =\{x, y, b\}(\text{Or1, Or2}) \implies =\{x, y, b\}(\text{Or1, Or2})].
\]
\text{proof.}
\text{eager proc.}
\text{eager seq 1 1 (incr : \text{Or1.incr}_x(); \text{Or1.incr}_y(); -
  \text{Or2.incr}_y(); \text{Or2.incr}_x(); :
  =\{x, y\}(\text{Or1, Or2}) \implies =\{x, y\}(\text{Or1, Or2})) :
  (=\{x, y, b\}(\text{Or1, Or2})).
\text{eager call eager_incr.}
\text{auto.}
\text{eager call eager_incr_x; first auto.}
\text{eager call eager_incr_y; first auto.}
\text{sim.}
\text{qed.}

\[\text{lemma} \text{ eager_incr_yx :}
\text{  eager}[\text{Or1.incr}_x(); \text{Or1.incr}_y();, \text{Or1.incr}_y -
  \text{Or2.incr}_y, \text{Or2.incr}_y(); \text{Or2.incr}_x(); :
  =\{x, y, b\}(\text{Or1, Or2}) \implies =\{x, y, b\}(\text{Or1, Or2}).
\]
\text{proof.}
\text{eager proc.}
\text{eager seq 1 1 (incr : \text{Or1.incr}_x(); \text{Or1.incr}_y(); -
  \text{Or2.incr}_y(); \text{Or2.incr}_x(); :
  =\{x, y\}(\text{Or1, Or2}) \implies =\{x, y\}(\text{Or1, Or2})) :
  (=\{x, y, b\}(\text{Or1, Or2})).
\text{eager call eager_incr.}
\text{first auto |}
\text{eager call eager_incr_y; first auto |}
\text{eager call eager_incr_x; first auto |}
\text{sim].
\text{qed.}
lemma eager_loop :
eager[Or1.incr_x(); Or1.incr_y();, Or1.loop -
Or2.loop, Or2.incr_y(); Or2.incr_x(); :
 ={n} /
 ={x, y, b}(Or1, Or2) =>
 ={res} /
 ={x, y, b}(Or1, Or2)].

proof.
eager proc.
swap(2) 2 2; wp.
eager while (incr : Or1.incr_x(); Or1.incr_y(); -
 Or2.incr_y(); Or2.incr_x(); :
 ={n} /
 ={x, y, b}(Or1, Or2) =>
 ={n} /
 ={x, y, b}(Or1, Or2)).
eager call eager_incr; first auto.
trivial.
swap(2) 2 2; wp.
eager if.
trivial.
mov=> &m b'; inline*; auto.
eager call eager_incr_x; first auto.
eager call eager_incr_y; first auto.
sim.
qed.

module type ADV(O : OR) = {
proc * main() : bool {O.incr_x O.incr_y O.incr_xy O.incr_yx O.loop}
}.

module G (Adv : ADV) = {
proc main1() : bool = {
var b : bool;
Or1.init();
Or1.incr_x(); Or1.incr_y();
b <@ Adv(Or1).main();
return b;
}
proc main2() : bool = {
var b : bool;
Or2.init();
b <@ Adv(Or2).main();
Or2.incr_y(); Or2.incr_x();
return b;
}
}

lemma eager_adv (Adv <: ADV{Or1, Or2}) :
eager[Or1.incr_x(); Or1.incr_y();, Adv(Or1).main -
 Adv(Or2).main, Or2.incr_y(); Or2.incr_x(); :
 ={x, y, b}(Or1, Or2) =>
 ={res} /
 ={x, y}(Or1, Or2)].

proof.
eager proc (incr : Or1.incr_x(); Or1.incr_y(); -
 Or2.incr_y(); Or2.incr_x(); :
 ={x, y}(Or1, Or2) =>
 ={x, y}(Or1, Or2))
 ={x, y, b}(Or1, Or2)).
eager call eager_incr; first auto.
trivial.
trivial.
apply eager_incr_x.
sim.
apply eager_incr_y.
sim.
apply eager_incr_xy.
sim.
apply eager_incr_yx.
sim.
apply eager_loop.
sim.
qed.

lemma G_Adv (Adv <: ADV{Or1, Or2}) :
equiv\[G(Adv).main1 ~ G(Adv).main2 :
   true ==>{res} /\ ={x, y}(Or1, Or2)].
proof.
proc.
seq 1 1 : (={x, y, b}(Or1, Or2)); first inline*; auto.
eager call (eager_adv Adv); first auto.
qed.

lemma G_Adv' (Adv <: ADV{Or1, Or2}) :
equiv\[G(Adv).main1 - G(Adv).main2 :
   true ==>{res} /\ ={x, y}(Or1, Or2)\].
proof.
proc.
seq 1 1 : (={x, y, b}(Or1, Or2)); first inline*; auto.
eager incr : Or1.incr_x(); Or1.incr_y(); -
   Or2.incr_y(); Or2.incr_x(); :
   ={x, y}(Or1, Or2) ==>({x, y}(Or1, Or2)) :
   ={x, y, b}(Or1, Or2)\]
eager call eager_incr; first auto.
auto.
eager proc incr (={x, y, b}(Or1, Or2));
[trivial | trivial | apply eager_incr_x | sim |
   apply eager_incr_y | sim | apply eager_incr_xy | sim |
   apply eager_incr_yx | sim | apply eager_loop | sim].
qed.

Syntax: eager proc. Turn a goal whose conclusion is an eager judgement into one whose
conclusion is a PRHL statement judgement in which the eager judgement’s procedures have been
replaced by their bodies. For example, if the current goal is

```
type variables: <none>

```
eager [ Or1.incr_x();, Or1.incr_y - Or2.incr_y, Or2.incr_x(
);
   Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} ==>
   Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}]
```

then running

```
eager proc.
```

produces the goal

```
type variables: <none>

```
&1 (left) : Or1.incr_y
&2 (right) : Or2.incr_y
pre = Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2}
Or1.incr_x() (1) Or2.y <= Or2.y + 1
Or1.y <= Or1.y + 1 (2) Or2.trace <= Or2.trace ++ [false]
(3) Or2.incr_x()

post = Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2}

Syntax: proc*. Turn a goal whose conclusion is an eager judgement into one whose conclusion is a PRHL statement judgement in which the eager judgement’s procedures are called, as opposed to being inlined. For example, if the current goal is

Type variables: <none>

```
eager[ Or1.incr_x();; Or1.incr_y();, Or1.incr_x ~ Or2.incr_x,
Or2.incr_y();; Or2.incr_x() ;
Or1.x{1} = Or2.x{2} \ /
Or1.y{1} = Or2.y{2} \ Or1.b{1} = Or2.b{2} ==>
Or1.x{1} = Or2.x{2} \ /
Or1.y{1} = Or2.y{2} \ Or1.b{1} = Or2.b{2}]
```

then running

```
proc*.
```

produces the goal

Type variables: <none>

```
&1 (left) : Or1.incr_x
&2 (right) : Or2.incr_x

pre =
  Or1.x{1} = Or2.x{2} /
  Or1.y{1} = Or2.y{2} /
  Or1.b{1} = Or2.b{2}

Or1.incr_x() (1) r <$> Or2.incr_x()
Or1.incr_y() (2) Or2.incr_y()
r <$> Or1.incr_x() (3) Or2.incr_x()

post =
  Or1.x{1} = Or2.x{2} /
  Or1.y{1} = Or2.y{2} /
  Or1.b{1} = Or2.b{2}
```

Syntax: eager call p. Here p is a proof term for an eager judgement. If the goal’s conclusion is a PRHL statement judgement whose programs’ suffixes match the left and right sides of the eager judgment, then consume those suffixes.

For example, if the current goal is

Type variables: <none>

```
&1 (left) : Or1.incr_xy
&2 (right) : Or2.incr_xy

pre = Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2}

Or1.incr_x() (1) Or2.incr_y()
Or1.incr_y() (2) Or2.incr_x()
```

CHAPTER 3. TACTICS

post = Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2}
then running

\texttt{eager call eager_incr.}

(see the statement of \texttt{eager_incr}, above) produces the goal

Type variables: <none>

&1 \texttt{(left)} : Or1.incr_xy
&2 \texttt{(right)} : Or2.incr_xy

pre = Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2}

post = Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2}

\textbf{Syntax: } \texttt{eager seq } n_1 \ n_2 \ (H : s_1 - s_2 : A \Rightarrow B) : C. 

Here, the goal’s conclusion must be a \texttt{pRHL} statement judgement whose left and right programs begin and end with \texttt{s_1} and \texttt{s_2}, respectively. A first subgoal is generated whose conclusion is the specified \texttt{pRHL} statement judgement, which is made available as \texttt{H} for the subsequent subgoals’s use. The \texttt{n_1} and \texttt{n_2} must be natural numbers saying how many statements from the part of the first program following \texttt{s_1} and from the beginning of the second program to put together with the \texttt{s_1} in a second \texttt{pRHL} statement judgement subgoal. The remaining statements are put together with the \texttt{s_i} in a third \texttt{pRHL} statement judgement subgoal. And there is a final subgoal whose conclusion is a \texttt{pRHL} statement judgment whose left and right sides are those remaining statements of the second program only.

For example, if the current goal is

Type variables: <none>

&1 \texttt{(left)} : Or1.incr_xy
&2 \texttt{(right)} : Or2.incr_xy

pre =
Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2} \ Or1.b{1} = Or2.b{2}

Or1.incr_x() \ (1) \ Or2.incr_x()
Or1.incr_y() \ (2) \ Or2.incr_y()
Or1.incr_x() \ (3) \ Or2.incr_y()
Or1.incr_y() \ (4) \ Or2.incr_x()

post =
Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2} \ Or1.b{1} = Or2.b{2}

then running

\texttt{eager seq 1 1 (incr : Or1.incr_xy(); Or1.incr_y(); - Or2.incr_y(); Or2.incr_x(); \ =\{x, y\}(Or1, Or2) ==> \{x, y\}(Or1, Or2)):}
\texttt{(\{x, y, b\}(Or1, Or2)).}

produces the goals
CHAPTER 3. TACTICS

Type variables: <none>

&1 (left) : Or1.incr_xy
&2 (right) : Or2.incr_xy

pre = Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2}  

Or1.incr_x() (1)  Or2.incr_y()  
Or1.incr_y() (2)  Or2.incr_x()  

post = Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2}  

and

Type variables: <none>

cr: equiv  

&1 (left) : Or1.incr_xy
&2 (right) : Or2.incr_xy

pre =  
Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2} \ Or1.b{1} = Or2.b{2}  

Or1.incr_x() (1)  Or2.incr_x()  
Or1.incr_y() (2)  Or2.incr_y()  
Or1.incr_x() (3)  Or2.incr_x()  

post =  
Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2} \ Or1.b{1} = Or2.b{2}  

and

Type variables: <none>

cr: equiv[

&1 (left) : Or1.incr_xy
&2 (right) : Or2.incr_xy

pre =  
Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2} \ Or1.b{1} = Or2.b{2}  

Or1.incr_x() (1)  Or2.incr_y()  
Or1.incr_y() (2)  Or2.incr_y()  
Or1.incr_y() (3)  Or2.incr_x()  

post =  
Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2} \ Or1.b{1} = Or2.b{2}  

and
and

Type variables: <none>

\begin{verbatim}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); - Or2.incr_y();;
  Or2.incr_x();;
  Or1.x{1} = Or2.x{2} \& Or1.y{1} = Or2.y{2} ==>
  Or1.x{1} = Or2.x{2} \& Or1.y{1} = Or2.y{2}]
\end{verbatim}

&1 (left) : Or2.incr_xy [programs are in sync]
&2 (right) : Or2.incr_xy

\begin{verbatim}
pre = ={Or2.b, Or2.y, Or2.x}

(1) Or2.incr_y()

post = ={Or2.b, Or2.y, Or2.x}
\end{verbatim}

**Syntax: eager if.** If the goal’s conclusion is a pRHL statement judgement whose left program consists of \( s_1 \) followed by a conditional, and whose right program consists of a conditional followed by \( s_2 \), reduce the goal to two subgoals using the \( s_i \) together with the “then” and “else” parts of the conditionals, along with auxiliary subgoals verifying that—even after running \( s_1 \)—the conditionals’ boolean expressions are equivalent.

For example, if the current goal is

Type variables: <none>

\begin{verbatim}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); - Or2.incr_y();;
  Or2.incr_x();;
  =\(n\) /\ Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} ==>
  =\(n\) /\ Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}]
\end{verbatim}

&1 (left) : Or1.loop
&2 (right) : Or2.loop

\begin{verbatim}
pre =
  =\(n\) /\ Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} /\ 0 < n{1}

Or1.incr_x() (1--) if (Or2.b) {
  (1.1) Or2.incr_x()
  (1--) } else {
  (1?1) Or2.incr_y()
  (1--) }

Or1.incr_y() (2--) Or2.incr_y()
if (Or1.b) {
  (3--) Or2.incr_x()
} else {
  (3--) Or1.incr_y()
  (3?1)
}

post =
  =\(n\) /\ Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} /\ (!Or1.b{1}) = !Or2.b{2}]
\end{verbatim}

then running
produces the goals

Type variables: <none>

\textbf{incr: equiv}\[\textsc{Or1.incr}_x() ; \textsc{Or1.incr}_y() ; - \textsc{Or2.incr}_y() ; \textsc{Or2.incr}_x() ; ;
\begin{align*}
&= \langle n \rangle \land \\
&\textsc{Or1.x}_1 = \textsc{Or2.x}_2 \land \\
&\textsc{Or1.y}_1 = \textsc{Or2.y}_2 \land \textsc{Or1.b}_1 = \textsc{Or2.b}_2 \Rightarrow \\
&= \langle n \rangle \land \\
&\textsc{Or1.x}_1 = \textsc{Or2.x}_2 \land \\
&\textsc{Or1.y}_1 = \textsc{Or2.y}_2 \land \textsc{Or1.b}_1 = \textsc{Or2.b}_2
\end{align*}
\]

\textbf{forall } &1 &2,
\begin{align*}
&(\langle n \rangle \land \textsc{Or1.x}_1 = \textsc{Or2.x}_2 \land \textsc{Or1.y}_1 = \textsc{Or2.y}_2 \land \textsc{Or1.b}_1 = \textsc{Or2.b}_2) \land \\
&0 < n \Rightarrow \textsc{Or1.b}_1 = \textsc{Or2.b}_2
\end{align*}

and

Type variables: <none>

\textbf{incr: equiv}\[\textsc{Or1.incr}_x() ; \textsc{Or1.incr}_y() ; - \textsc{Or2.incr}_y() ; \textsc{Or2.incr}_x() ; ;
\begin{align*}
&= \langle n \rangle \land \\
&\textsc{Or1.x}_1 = \textsc{Or2.x}_2 \land \\
&\textsc{Or1.y}_1 = \textsc{Or2.y}_2 \land \textsc{Or1.b}_1 = \textsc{Or2.b}_2 \Rightarrow \\
&= \langle n \rangle \land \\
&\textsc{Or1.x}_1 = \textsc{Or2.x}_2 \land \\
&\textsc{Or1.y}_1 = \textsc{Or2.y}_2 \land \textsc{Or1.b}_1 = \textsc{Or2.b}_2
\end{align*}
\]

\textbf{forall } &m2 (b1 : bool),
\begin{align*}
&\textbf{hoare}\[\textsc{Or1.incr}_x() ; \textsc{Or1.incr}_y() ; ;
\begin{align*}
&= (\langle n \rangle \land \textsc{Or1.x}_1 = \textsc{Or2.x}_2 \land \textsc{Or1.y}_1 = \textsc{Or2.y}_2 \land \textsc{Or1.b}_1 = \textsc{Or2.b}_2) \land \\
&0 < n \land \\
&\textsc{Or1.b} = b1 \Rightarrow \textsc{Or1.b} = b1
\end{align*}
\]

and

Type variables: <none>

\textbf{incr: equiv}\[\textsc{Or1.incr}_x() ; \textsc{Or1.incr}_y() ; - \textsc{Or2.incr}_y() ; \textsc{Or2.incr}_x() ; ;
\begin{align*}
&= \langle n \rangle \land \\
&\textsc{Or1.x}_1 = \textsc{Or2.x}_2 \land \\
&\textsc{Or1.y}_1 = \textsc{Or2.y}_2 \land \textsc{Or1.b}_1 = \textsc{Or2.b}_2 \Rightarrow \\
&= \langle n \rangle \land \\
&\textsc{Or1.x}_1 = \textsc{Or2.x}_2 \land \\
&\textsc{Or1.y}_1 = \textsc{Or2.y}_2 \land \textsc{Or1.b}_1 = \textsc{Or2.b}_2
\end{align*}
\]

&1 (left ) : Or1.loop
&2 (right ) : Or2.loop

\textbf{pre} =
\begin{align*}
&= (\langle n \rangle \land \textsc{Or1.x}_1 = \textsc{Or2.x}_2 \land \textsc{Or1.y}_1 = \textsc{Or2.y}_2 \land \textsc{Or1.b}_1 = \textsc{Or2.b}_2) \land \\
&0 < n \land \\
&\textsc{Or1.b}(hr) = \text{true}
\end{align*}

\textsc{Or1.incr}_x() \quad \begin{array}{l}(1) \quad \textsc{Or2.incr}_x() \end{array}
Or1.incr_y() (2) Or2.incr_y()
Or1.incr_x() (3) Or2.incr_x()

post =
={n} /
Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2} /\ (!Or1.b{1}) = !Or2.b{2}

and

Type variables: <none>

incr: equiv[ Or1.incr_x(); Or1.incr_y(); ~ Or2.incr_y();; Or2.incr_x(); ;
=({n} /\ Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2} =>
=({n} /\ Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2})]

&1 (left) : Or1.loop
&2 (right) : Or2.loop

pre =
=({n} /\ Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2}) /\ 0 < n{1}) /\ Or1.b(hr) = false
Or1.incr_x() (1) Or2.incr_y()
Or1.incr_y() (2) Or2.incr_y()
Or1.incr_y() (3) Or2.incr_x()

post =
=({n} /\ Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} /\ (!Or1.b{1}) = !Or2.b{2}

Syntax: eager while (H : s1 - s2 : A => B). Like eager if, but working with while loops instead of conditionals and featuring an explicit, named pRHL statement judgement involving the si, available as H to the subgoals. The subgoal involving the bodies of the while loops uses B as its invariant. There is also a subgoal whose conclusion is a pRHL statement judgement both of whose sides are the body of the second program’s while loop.

For example, if the current goal is

Type variables: <none>

&1 (left) : Or1.loop
&2 (right) : Or2.loop

pre =
=({n} /\ Or1.x{1} = Or2.x{2} /\ Or1.y{1} = Or2.y{2} /\ Or1.b{1} = Or2.b{2})
Or1.incr_x() (1----) while (0 < n) {
(1.1--) if (Or2.b) {
(1.1.1) Or2.incr_x()
(1.1--) } else {
(1.1?1) Or2.incr_y()
(1.1--) }
\textbf{CHAPTER 3. TACTICS}

\begin{itemize}
  \item (1.2--) \text{Or2.b} \leftarrow \neg \text{Or2.b}
  \item (1----) \}
  \text{Or1.incr_y()}
  \item (2----) \text{Or2.incr_y()}
  \item (3----) \text{Or2.incr_x()}
  \item (3.1--) \text{Or1.incr_x()}
  \item (3.1.1) \text{Or1.incr_y()}
  \item (3.1--) \text{Or1.incr_y()}
  \item (3.1--) \text{Or1.b \leftarrow \neg \text{Or1.b}}
  \item (3.2--) \}
\end{itemize}

\text{post = (} \neg \text{Or1.b(1)} = \neg \text{Or2.b(2)} \}\land
\text{Or1.x(1) = Or2.x(2)} \land
\text{Or1.y(1) = Or2.y(2)} \land \text{Or1.b(1) = Or2.b(2)}\}

\text{then running}

\textbf{eager while} \text{(incr : Or1.incr_x(); Or1.incr_y(); \text{=} Or2.incr_y(); Or2.incr_x(); : \text{=} \{n\} \land \text{=} \{x, y, b\}(\text{Or1, Or2}) =>
\text{=} \{n\} \land \text{=} \{x, y, b\}(\text{Or1, Or2}).

\text{produces the goals}

\text{Type variables: <none>}

\&1 \text{ (left) : Or1.loop}
\&2 \text{ (right) : Or2.loop}

\text{pre = (} \{n\} \land
\text{Or1.x(1) = Or2.x(2)} \land
\text{Or1.y(1) = Or2.y(2)} \land \text{Or1.b(1) = Or2.b(2)}\}
\text{Or1.incr_x()}
\text{Or1.incr_y()}

\text{post = (} \{n\} \land
\text{Or1.x(1) = Or2.x(2)} \land
\text{Or1.y(1) = Or2.y(2)} \land \text{Or1.b(1) = Or2.b(2)}\}

\&1 \&2 , \{n\} \land
\text{Or1.x(1) = Or2.x(2)} \land
\text{Or1.y(1) = Or2.y(2)} \land \text{Or1.b(1) = Or2.b(2)} =>

\text{Type variables: <none>}

\text{incr: equiv[ Or1.incr_x(); Or1.incr_y(); \neg Or2.incr_y();
\text{Or2.incr_x(); : \text{=} \{n\} \land
\text{Or1.x(1) = Or2.x(2)} \land
\text{Or1.y(1) = Or2.y(2)} \land \text{Or1.b(1) = Or2.b(2)} =>
\text{=} \{n\} \land
\text{Or1.x(1) = Or2.x(2)} \land
\text{Or1.y(1) = Or2.y(2)} \land \text{Or1.b(1) = Or2.b(2)}]

\text{forall \&1 \&2,}
\text{=} \{n\} \land
\text{Or1.x(1) = Or2.x(2)} \land
\text{Or1.y(1) = Or2.y(2)} \land \text{Or1.b(1) = Or2.b(2)} =>}
0 < n(1) = 0 < n(2)

and

Type variables: <none>

incr: equiv

0 < n(1) = 0 < n(2)

and

Type variables: <none>

incr: equiv

0 < n(1) = 0 < n(2)
```plaintext
(1.1) Or2.incr_x()
(1--) } else {
(1?1) Or2.incr_y()
(1--) }
(2--) Or2.b <- !Or2.b

post = ={n, Or2.b, Or2.y, Or2.x}
```

**Syntax:** `eager proc (H : s₁ - s₂ : A => B) C | eager H C`. This is the form of `eager proc` that applies when the procedures $M.p$ and $N.q$ are abstract. The second variant is where the specified PRHL statement judgement has already been introduced by another `eager` tactic. There must be a module restriction saying that the abstract procedures can’t directly interfere with the global variables on which the $s_i$ depend. Subgoals are generated for each pair of “oracles” the abstract procedures are capable of calling, i.e., for each of the procedures that may read/write the global variables used by the $s_i$.

For example, if the current goal is

```
Type variables: <none>
Adv: ADV{Or1, Or2}

```

```
eager[ Or1.incr_x();; Or1.incr_y();, Adv(Or1).main ~ Adv(Or2).main, Or2.incr_y();; Or2.incr_x(); :
Or1.x(1) = Or2.x(2) \ /
Or1.y(1) = Or2.y(2) \ /
Or1.b(1) = Or2.b(2) ==>
={res} \ /
Or1.x(1) = Or2.x(2) \ /
Or1.y(1) = Or2.y(2)]
```

then running

```
eager proc (incr : Or1.incr_x(); Or1.incr_y(); ~
Or2.incr_y(); Or2.incr_x(); :
={x, y}(Or1, Or2) ==>
={x, y}(Or1, Or2))
```

produces the goals

```
Type variables: <none>
Adv: ADV{Or1, Or2}

```

```
&1 (left) : Top.
&2 (right) : Top.

pre = Or1.x(1) = Or2.x(2) \ /
Or1.y(1) = Or2.y(2)
```

```
Or1.incr_x()
(1) Or2.incr_y()
Or1.incr_y()
(2) Or2.incr_x()
post = Or1.x(1) = Or2.x(2) \ /
Or1.y(1) = Or2.y(2)
```

and

```
Type variables: <none>
Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x(); Or1.incr_y(); ~ Or2.incr_y();;
Or2.incr_x(); :
Or1.x(1) = Or2.x(2) \ /
Or1.y(1) = Or2.y(2) ==>
Or1.x(1) = Or2.x(2) \ /
Or1.y(1) = Or2.y(2)]
```
forall \&1 \&2,
Or1.x{1} = Or2.x{2} \land
Or1.y{1} = Or2.y{2} \land Or1.b{1} = Or2.b{2} =>
true \land
Or1.x{1} = Or2.x{2} \land
Or1.y{1} = Or2.y{2} \land Or1.b{1} = Or2.b{2}

and

Type variables: <none>

Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); - Or2.incr_y();
            Or2.incr_x(); ;
            Or1.x{1} = Or2.x{2} \land Or1.y{1} = Or2.y{2} =>
            Or1.x{1} = Or2.x{2} \land Or1.y{1} = Or2.y{2}]

forall \&1 \&2,
={res} \land
={glob Adv} \land
Or1.x{1} = Or2.x{2} \land
Or1.y{1} = Or2.y{2} \land Or1.b{1} = Or2.b{2} =>
={res} \land Or1.x{1} = Or2.x{2} \land Or1.y{1} = Or2.y{2}

and

Type variables: <none>

Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); - Or2.incr_y();
            Or2.incr_x();
            Or1.x{1} = Or2.x{2} \land Or1.y{1} = Or2.y{2} =>
            Or1.x{1} = Or2.x{2} \land Or1.y{1} = Or2.y{2}]

eager[ Or1.incr_x();; Or1.incr_y();, Or1.incr_x - Or2.incr_x,
       Or2.incr_x();; Or2.incr_y();
       true \land
       Or1.x{1} = Or2.x{2} \land
       Or1.y{1} = Or2.y{2} \land Or1.b{1} = Or2.b{2} =>
       ={res} \land
       Or1.x{1} = Or2.x{2} \land
       Or1.y{1} = Or2.y{2} \land Or1.b{1} = Or2.b{2}]

and

Type variables: <none>

Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x();; Or1.incr_y(); - Or2.incr_y();
            Or2.incr_x();
            Or1.x{1} = Or2.x{2} \land Or1.y{1} = Or2.y{2} =>
            Or1.x{1} = Or2.x{2} \land Or1.y{1} = Or2.y{2}]

pre = ={Or2.b, Or2.y, Or2.x}
Or2.incr_x - Or2.incr_x

post = ={res, Or2.b, Or2.y, Or2.x}

and

Type variables: <none>
CHAPTER 3. TACTICS

Adv: ADV\{Or1, Or2\}
incr: equiv\[ Or1.incr\_x();; Or1.incr\_y(); - Or2.incr\_y();; Or2.incr\_x();; Or1.x\{1\} = Or2.x\{2\} \wedge Or1.y\{1\} = Or2.y\{2\} \Rightarrow
   Or1.x\{1\} = Or2.x\{2\} \wedge Or1.y\{1\} = Or2.y\{2\}\]

\texttt{eager}\[ Or1.incr\_x();; Or1.incr\_y();, Or1.incr\_y - Or2.incr\_y,
   Or2.incr\_y();; Or2.incr\_x();;
   true /\ Or1.x\{1\} = Or2.x\{2\} /\ Or1.y\{1\} = Or2.y\{2\} \Rightarrow
   ={res} /\ Or1.x\{1\} = Or2.x\{2\} /\ Or1.y\{1\} = Or2.y\{2\}\]

and

Type variables: <none>

Adv: ADV\{Or1, Or2\}
incr: equiv\[ Or1.incr\_x();; Or1.incr\_y(); - Or2.incr\_y();; Or2.incr\_x();; Or1.x\{1\} = Or2.x\{2\} \wedge Or1.y\{1\} = Or2.y\{2\} \Rightarrow
   Or1.x\{1\} = Or2.x\{2\} \wedge Or1.y\{1\} = Or2.y\{2\}\]

pre = ={Or2.b, Or2.y, Or2.x}

Or2.incr\_y - Or2.incr\_y

post = ={res, Or2.b, Or2.y, Or2.x}

and

Type variables: <none>

Adv: ADV\{Or1, Or2\}
incr: equiv\[ Or1.incr\_x();; Or1.incr\_y(); - Or2.incr\_y();; Or2.incr\_x();; Or1.x\{1\} = Or2.x\{2\} \wedge Or1.y\{1\} = Or2.y\{2\} \Rightarrow
   Or1.x\{1\} = Or2.x\{2\} \wedge Or1.y\{1\} = Or2.y\{2\}\]

\texttt{eager}\[ Or1.incr\_x();; Or1.incr\_y();, Or1.incr\_xy - Or2.incr\_xy,
   Or2.incr\_y();; Or2.incr\_x();;
   true /\ Or1.x\{1\} = Or2.x\{2\} /\ Or1.y\{1\} = Or2.y\{2\} \Rightarrow
   ={res} /\ Or1.x\{1\} = Or2.x\{2\} /\ Or1.y\{1\} = Or2.y\{2\}\]

and

Type variables: <none>

Adv: ADV\{Or1, Or2\}
incr: equiv\[ Or1.incr\_x();; Or1.incr\_y(); - Or2.incr\_y();; Or2.incr\_x();; Or1.x\{1\} = Or2.x\{2\} \wedge Or1.y\{1\} = Or2.y\{2\} \Rightarrow
   Or1.x\{1\} = Or2.x\{2\} \wedge Or1.y\{1\} = Or2.y\{2\}\]

pre = ={Or2.b, Or2.y, Or2.x}
Or2.incr_xy - Or2.incr_xy

post = =\{res, Or2.b, Or2.y, Or2.x\}

and

Type variables: <none>

Adv: ADV\{Or1, Or2\}
incr: equiv [ Or1.incr_x();; Or1.incr_y();; - Or2.incr_y();;
Or2.incr_x(); :
Or1.x{1} = Or2.x{2} \ / Or1.y{1} = Or2.y{2} ==>
Or1.x{1} = Or2.x{2} \ / Or1.y{1} = Or2.y{2}]

eager [ Or1.incr_x();; Or1.incr_y();, Or1.incr_yx - Or2.incr_yx,
Or2.incr_y();; Or2.incr_x(); :
true \ /
Or1.x{1} = Or2.x{2} \ /
Or1.y{1} = Or2.y{2} \ / Or1.b{1} = Or2.b{2} ==>
=\{res\} \ /
Or1.x{1} = Or2.x{2} \ /
Or1.y{1} = Or2.y{2} \ / Or1.b{1} = Or2.b{2}]

and

Type variables: <none>

Adv: ADV\{Or1, Or2\}
incr: equiv [ Or1.incr_x();; Or1.incr_y();; - Or2.incr_y();;
Or2.incr_x(); :
Or1.x{1} = Or2.x{2} \ / Or1.y{1} = Or2.y{2} ==>
Or1.x{1} = Or2.x{2} \ / Or1.y{1} = Or2.y{2}]

pre = ={Or2.b, Or2.y, Or2.x}

Or2.incr_yx - Or2.incr_yx

post = =\{res, Or2.b, Or2.y, Or2.x\}

and

Type variables: <none>

Adv: ADV\{Or1, Or2\}
inrr: equiv [ Or1.incr_x();; Or1.incr_y();; - Or2.incr_y();;
Or2.incr_x(); :
Or1.x{1} = Or2.x{2} \ / Or1.y{1} = Or2.y{2} ==>
Or1.x{1} = Or2.x{2} \ / Or1.y{1} = Or2.y{2}]

eager [ Or1.incr_x();; Or1.incr_y();, Or1.loop - Or2.loop,
Or2.incr_y();; Or2.incr_x(); :
=\{n\} \ /
Or1.x{1} = Or2.x{2} \ /
Or1.y{1} = Or2.y{2} \ / Or1.b{1} = Or2.b{2} ==>
=\{res\} \ /
Or1.x{1} = Or2.x{2} \ /
Or1.y{1} = Or2.y{2} \ / Or1.b{1} = Or2.b{2}]

and

Type variables: <none>
Adv: ADV{Or1, Or2}
incr: equiv[ Or1.incr_x(); Or1.incr_y(); Or2.incr_y();
Or2.incr_x(); :
Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2} =>
Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2}]

pre = ={n, Or2.b, Or2.y, Or2.x}
Or2.loop ~ Or2.loop

post = ={res, Or2.b, Or2.y, Or2.x}

Syntax: eager \( H : s_1 - s_2 : A \Rightarrow B \) : C. Reduces a goal whose conclusion is a pRHL
statement judgement of the form
\[ \text{equiv}[s_1 t_1 - t_2 s_2 : P \Rightarrow Q], \]
to an eager judgement, along with a subgoal for the specified pRHL statement judgement \( H \),
plus an auxiliary goal.

For example, if the current goal is

Type variables: <none>
Adv: ADV{Or1, Or2}

&1 (left) : G(Adv).main1
&2 (right) : G(Adv).main2

pre =
Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2} \ Or1.b{1} = Or2.b{2}
Or1.incr_x()
Or1.incr_y()
b @$Adv(Or2).main()$

post = ={b} \ Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2}

then running

eager (incr : Or1.incr_x(); Or1.incr_y(); -
Or2.incr_y(); Or2.incr_x(); :
=\langle x, y\rangle(Or1, Or2) ==> =\langle x, y\rangle(Or1, Or2)) :
=\langle x, y, b\rangle(Or1, Or2)).

produces the goals

Type variables: <none>
Adv: ADV{Or1, Or2}

&1 (left) : G(Adv).main1
&2 (right) : G(Adv).main2

pre = Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2}
Or1.incr_x()
Or1.incr_y()

post = Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2}
and

Type variables: <none>

Adv: ADV(Or1, Or2)
incr: equiv[ Or1.incr_x(); Or1.incr_y(); - Or2.incr_y();
       Or2.incr_x(); :
       Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2} ==>
       Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2}]

&1 (left) : G(Adv).main1
&2 (right) : G(Adv).main2

pre =
   Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2} \ Or1.b{1} = Or2.b{2}

post =
   Or1.b{1} = Or2.b{2} \ Or1.y{1} = Or2.y{2} \ Or1.x{1} = Or2.x{2}

and

Type variables: <none>

Adv: ADV(Or1, Or2)
incr: equiv[ Or1.incr_x(); Or1.incr_y(); - Or2.incr_y();
       Or2.incr_x(); :
       Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2} ==>
       Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2}]

eager[ Or1.incr_x(); Or1.incr_y();, Adv(Or1).main ~
       Adv(Or2).main, Or2.incr_y(); Or2.incr_x(); :
       Or1.x{1} = Or2.x{2} \ Or1.y{1} = Or2.y{2} \ Or1.b{1} = Or2.b{2} ==>
       (Or1.y{1} = Or2.y{2} \ Or1.x{1} = Or2.x{2}) / ={res}]}
Chapter 4

Structuring Specifications and Proofs

4.1 Theories
4.2 Sections
Chapter 5

EasyCrypt Library
Chapter 6

Advanced Features and Usage
Chapter 7

Examples

7.1 Hashed ElGamal

7.2 BR93

We’ll work through [BR93].
Bibliography


Index of Tactics

admit, 58
algebra, 68
alias, 125
apply, 60
assumption, 48
auto, 171
byequiv, 149
byphoare, 156
bypr, 162
call, 93
case, 64
case-pl, 147
cfold, 126
change, 58
clear, 47
closing goals, 71
congr, 53
conseq, 136
cut, 60
done, 56
eager, 182
elim, 67
elim*, 166
exact, 61
exfalso, 170
exists, 50
exists*, 164
failure recovery, 69
fel, 177
fission, 122
fusion, 123
generalization, 46
goal selection, 70
have, 59
hoare, 169
idtac, 47
if, 86
inline, 106
introduction, 41
kill, 128
left, 48
move, 47
phoare split, 169
pose, 59
proc, 72
progress, 57
rcondf, 113
rcondt, 108
reflexivity, 48
rewrite, 62
right, 49
rnd, 83
seq, 79
sequence, 69
sequence with branching, 69
sim, 173
simplify, 56
skip, 79
smt, 57
sp, 81
split, 50
splitwhile, 119
subst, 54
swap, 103
symmetry, 130
tactic repetition, 70
transitivity, 131
trivial, 55
unroll, 117
while, 89
wp, 82
List of Corrections

Note: Add explanation of new replace tactic, which uses statement patterns. 38
Note: Need explanation of how a proof term may be used in forward reasoning. 40
Note: Can it be used in forward reasoning? 40
Note: In forward reasoning they aren’t equivalent—why? 40
Note: Be a bit more detailed about what this tactic does? 56
Note: Is this the right place to define “convertible”? 57
Note: Describe progress options. 57
Note: Describe failure states of prover selection. 57
Note: Describe dbhint options. 58
Note: Make this a pragma? 58
Note: Missing description of algebra. 68
Fatal: Why is the second subgoal pruned? (Compare with first and second variants, where the corresponding subgoal isn’t pruned.) 154
Fatal: Why is the second subgoal pruned? 155
Fatal: It’s confusing how the third goal has been simplified, but not pruned. 159
Fatal: It’s confusing how the third goal has been simplified, but not pruned. 161
Fatal: Update waiting for overhaul of pHL. 169
Fatal: Update waiting for overhaul of pHL. 169
Fatal: Perhaps need other examples? 171
Fatal: Need better description of when the tactic is applicable and how the tactic works! 171