EasyCrypt Reference Manual

Version 1.x — November 24, 2014

1 Tactics

EasyCrypt comes with a proof engine that allows to state and prove properties about programs written in the various languages. Proofs are built interactively by applying tactics, that transform a current proof goal into a (possibly empty) set of subgoals such that the conjunction of the subgoals implies the original goal. This process is repeated, starting from the theorem statement, up to the point where all the subgoals correspond to general axioms or premises of the theorem.

In this chapter, we describe this proof engine in general before listing and describing the existing tactics for various fragments of EasyCrypt's underlying logic.

2 The proof engine

The proof engine deals with judgments or goals of the form $\epsilon; \Gamma \vdash \phi$ where ϵ is the (global) environment, Γ is the context (a set of local facts) and ϕ is the formula we want to prove. Here is an example of such a judgment:

$$\mathsf{Int}; x, y, z : \mathsf{int}, x \leq y \vdash x + z \leq y + z.$$

It states that in the environment (Int) solely composed of the theory of integers, having three local variables x, y, z of type int along with the fact $x \leq y$ (the context Γ), we are interested in proving $x + z \leq y + z$.

On top on this, a set of deduction rules is given. They describe how one can derive a judgment $\epsilon; \Gamma \vdash \phi$ given that a set of prerequisites (or premises) are fulfilled. The general form of such a rule is given as follow:

$$\frac{A_1 \cdots A_n}{\epsilon; \Gamma \vdash \phi}$$

It has to be read as: *given that* $A_1 \cdots A_n$ *are derivable, then so is* $\epsilon, \Gamma \vdash \phi$. We give three examples of such deduction rules:

$$\frac{\epsilon; \Gamma \vdash \phi_1 \quad \epsilon; \Gamma \vdash \phi_1 \Rightarrow \phi_2}{\epsilon; \Gamma \vdash \phi_2}\ \text{LEFT=MP} \qquad \frac{\epsilon; \Gamma, \phi_1 \vdash \phi_2}{\epsilon; \Gamma \vdash \phi_1 \Rightarrow \phi_2}\ \text{LEFT=}\Rightarrow\text{-I} \qquad \frac{}{\epsilon; \Gamma, \phi, \Delta \vdash \phi}\ \text{LEFT=Ax}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\epsilon; b_1, b_2 : \mathsf{bool}, b_1 \Rightarrow b_2, b_1 \vdash b_1 \Rightarrow b_2} \quad \dfrac{}{\epsilon; b_1, b_2 : \mathsf{bool}, b_1 \Rightarrow b_2, b_1 \vdash b_1}}{\epsilon; b_1, b_2 : \mathsf{bool}, b_1 \Rightarrow b_2, b_1 \vdash b_2}}{\epsilon; b_1, b_2 : \mathsf{bool}, b_1 \Rightarrow b_2 \vdash b_1 \Rightarrow b_2}}{\epsilon; b_1, b_2 : \mathsf{bool} \vdash (b_1 \Rightarrow b_2) \Rightarrow b_1 \Rightarrow b_2}$$

Figure 1: Proof tree of $\epsilon; b_1, b_2 : \mathsf{bool} \vdash (b_1 \Rightarrow b_2) \Rightarrow b_1 \Rightarrow b_2$

The first, the *modus ponens*, states that one can derive $\epsilon; \Gamma \vdash \phi_2$ given that $\epsilon; \Gamma \vdash \phi_1 \Rightarrow \phi_2$ and $\epsilon; \Gamma \vdash \phi_1$ are derivable. The next provides a way for deriving $\phi_1 \Rightarrow \phi_2$ from a derivation of ϕ_2, but with a context augmented by ϕ_1. The last states that $\epsilon; \Gamma, \phi, \Delta \vdash \phi$ is derivable as-is.

Combining these deduction rules, it is possible to build a tree rooted by a judgment $\epsilon; \Gamma \vdash \phi$ and with leaves composed of deduction rules with no premises (as the third one in the previous example). Such a tree forms a proof of $\epsilon; \Gamma \vdash \phi$. For instance, Figure 1 gives a proof of

$$\epsilon; b_1, b_2 : \mathsf{bool} \vdash (b_1 \Rightarrow b_2) \Rightarrow b_1 \Rightarrow b_2$$

The EasyCrypt proof engine helps the user build such proofs. At each step of the proof building, the system presents to the user the set of goals that have to be proved. The user can then *apply* a tactic to one of them, each tactic corresponding to a deduction rule. If the conclusion of the rule corresponding to the applied tactic matches the goal to which it is applied, the proof engine replaces it with the set of the premises of the applied rule - the subgoals. This application may generate no, one or several subgoals depending on the rule. This process is repeated iteratively, up to the point where no goals remain.

2.1 Ambient logic

◎ idtac

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

◎ move | move: $\pi_1 \cdots \pi_n$

Syntax: move. Does nothing, equivalent to idtac (p. 2). This form is mainly used in conjonction with an introduction pattern (see Section **??**), e.g. move⇒ $\iota_1 \cdots \iota_n$.

Syntax: move: $\pi_1 \cdots \pi_n$. Generalize the patterns $\pi_1, \cdots, \pi_n$, starting from π_n and going back.

◎ clear $x_1 \cdots x_n$

Clear the local variables an hypotheses $x_1 \cdots x_n$ from the local context. Fail if the remaining ones depend on the cleared ones.

◎ done

Apply trivial (p. 5) and fail if the goal the not closed.

◎ apply (p : proof−term)

If p is a proof-term for the pattern (formula)

$$\textit{forall}\ (x1 : t1)\ \ldots\ (xn : tn),\ A1 \to \ldots \to An \to B$$

then apply tries to match B with the current G. If the match succeeds and leads to the full instantiation of the pattern, then the goal is replaced, after instantiation, with the n subgoals A1, ..., An.

◎ exact (p : proofterm)

Equivalent to by apply (p : proofterm), i.e. apply the given proof-term and the try to close the goals with trivial - failing if not all goals can be closed.

◎ assumption

Search in the context for a hypothesis that is convertible to the goal and apply. Fail otherwise.

◎ pose x := π

Search for the first subterm of the goal matching p and leading to the full instantiation of the pattern. Then introduce, after instantiation, the local definition x := π and abstract all instantiated occurrences of p in the goal by x. An occurence selector can be used (see rewrite (p. 3)).

◎ cut ι: ϕ

Logical cut. Generate two subgoals: one for the cut formula ϕ, and one for $\phi \Rightarrow G$ where G is the current goal. Moreover, the intro-pattern ι is applied to the second subgoal.

◎ rewrite $\pi_1 \cdots \pi_n$

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

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

The two last forms can be prefixed by a direction indicator (the sign −), followed by an occurrence selector ({i1 ... in}), followed (for proof-terms only) by a repetition marker (!, ?, n! or n?). All these prefixes are optional.
Depending on the form of π, rewrite π does the following:

- For //, /=, and //=, see intros.
- If rw is a proof-term for the pattern

$$\textit{forall}\ (x1 : t1)\ \ldots\ (xn : tn),\ A1 \to \ldots \to An \to f1 = f2$$

then rewrite searches for the first subterm of the goal matching f1 and resulting in the full instantiation of the pattern. It then replaces, after instantiation of the pattern, all the occurrences of f1 by f2 in the goal, and creates n new subgoals for the Ai's. If no subterms of the goal match f1 or if the pattern cannot be fully instantiated by matching, the tactic fails. The tactic works the same if the pattern ends by f1 $\Leftrightarrow$ f2. If the direction indicator $-$ is given, rewrite works in the reverse direction, searching for a match of f2 and then replacing all occurrences of f2 by f1.

- If rw is a /-prefixed pattern of the form (o p1 ... pn), with o a defined symbol, then rewrite searches for the first subterm of the goal matching (o p1 ... pn) and resulting in the full instantiation of the pattern. It then replaces, after instantiation of the pattern, all the occurrences of (o p1 ... pn) by the $\beta\delta$ head-normal form of (o p1 ... pn), where the δ-reduction is restricted to subterms headed by the symbol o. If no subterms of the goal match (o p1 ... pn) 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 $\beta\delta_{\text{o}}$ head-normal of (o p1 ... pn) and then replacing all occurrences of this head-normal form with (o p1 ... pn).

The occurrence selector {i1 ... in} restricts which occurrences of the matching pattern are replaced in the goal. If given, only the i1-th, ..., in-th ones are replaced (considering that the goal is traversed in DFS mode). Note that this selection applies after the matching has been done.

Repetition markers allow the repetition of the same rewriting. For instance, rewrite π leads to do! rewrite π. See do for more information.

Last, rewrite rw_1 ... rw_n is equivalent to rewrite rw_1; ...; rewrite rw_n.

◎ subst | subst x

Search for the first equation of the form x = f or f = x in the context and replace all the occurrences of x by f everywhere in the context and the goal before clearing it. If no identifier is given, repeatedly apply the tactic to all identifiers for which such an equation exists.

◎ split

Break an intrinsically conjunctive goal into its component subgoals. 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 implication,
- replace a goal of the form f1 $\wedge$ f2 by the two subgoals for f1 an f2. The same applies for a goal of the form f1 && f2,
- replace an equality between n-tuples by n equalities on their components.

◎ left

Reduce a disjunctive goal to its left member.

◎ right

Reduce a disjunctive goal to its left member.

◎ case ϕ

Do an excluded-middle case analysis on ϕ, substituting ϕ in the goal.

◎ elim/ϕ $\pi_1 \cdots \pi_n$

Apply the elimination principle ϕ to the top assumption after having generalizing $\pi_1 \cdots \pi_n$.

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

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

◎ progress | progress τ

Break the goal into multiple *simpler* ones by repeatedly applying split, subst and move⇒. The tactic τ given to progress is tentatively applied after each step.

◎ reflexivity

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

◎ congr

Replace a goal of the form f t1 ... tn = f u1 ... un with the subgoals ti = ui for all i. Subgoals solvable by reflexivity are automatically closed.

◎ trivial

Try to solve the goal by using a mixture of low-level tactics. This tactic is called by the intro-pattern //.

◎ smt

Try to solve the goal using SMT solvers. The goal is sent along with all the lemmas proved so far plus the local hypotheses.

◎ admit

Close the current goal by admitting it.

2.2 Tacticals

Tactics can be combined together, composed and modified by tacticals. Tacticals do not correspond to any deduction rule but make the proof process smoother, and sometimes permit the reuse of proofs with similar patterns, but where the fine minutiae might differ.

◎ t1; t2

Execute t1 and then t2 on all the subgoals generated by t1.

◎ try t

Execute the tactic t if it succeeds; do noting 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 a bunch of smt.

◎ do! t

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

Variants

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

◎ t1; first t2

Apply the tactic t1, then apply t2 on the first subgoal generated by t1. An error is produced if no subgoals have been generated by t1.

Variants

t1; first n t2	apply t2 on the first n subgoals generated by t1
t1; last t2	apply t2 on the last subgoal generated by t1
t1; last n t2	apply t2 on the last n subgoals generated by t1
t; first n last	reorder the subgoals generated by t, moving the first n to the end of the list

◎ by t

Apply the tactic t and try to close all the generated subgoals using trivial (p. 5). Fail if not all subgoals can be closed.

2.3 Program Logics

Judgments in the program logics may refer to procedures or statements. Whenever the context allows both, we use c (or c) to denote programs, using f (or f) when only judgments on procedures are allowed by the context, and s (or s) when only judgments on statements are allowed.

EasyCrypt includes three different program logics:

- pRHL, or probabilistic relational Hoare logic, with judgments of the form

$$\{P\}\ c_1 \sim c_2\ \{Q\}$$

  where c_1 and c_2 are programs, and P and Q are relations on memories.

- pHL, or probabilistic Hoare logic, with judgments of the form

$$\{P\}\ c\ \{Q\} \diamond \delta$$

  where c is a program, P and Q are predicates on memories, $\diamond \in \{\leq, \geq, =\}$ is a comparison relation and δ is a real-valued expression, evaluated in the initial memory.

- HL, or (possibilistic) Hoare logic, with judgments of the form

$$\{P\}\ c\ \{Q\}$$

  where c is a program, and P and Q are predicates on memories.

When c is a procedure, preconditions (P above) operate on memories extended with a special arg location that refers to the procedure's arguments, and postcondition (Q above) operate on memories extended with a special res location that refers to the procedure's return value.

In the following, given a relation R, we denote with R^{-1} its inverse relation (that is, $m_1\ R\ m_2 \Leftrightarrow m_2\ R^{-1}\ m_1$).

We denote with $\diamond^{\uparrow}$ the function defined by

$$.^{\uparrow} = \begin{cases} = & \mapsto & \Leftrightarrow \\ \leq & \mapsto & \Leftarrow \\ \geq & \mapsto & \Rightarrow \end{cases}$$

Given a predicate P, we denote with $P\langle 1\rangle$ (resp. $P\langle 2\rangle$) the relation defined by $m_1\ P\langle 1\rangle\ m_2 \Leftrightarrow P\ m_1$ (resp. $m_1\ P\langle 2\rangle\ m_2 \Leftrightarrow P\ m_2$). We lift logical connectors to predicates and relations over memories in the natural way.

TODO: define <spec>, <**lemma**>, <prhl>, <phl>, <hl>.

Reasoning on Specifications

◎ symmetry

Syntax: symmetry. In pRHL, swaps the two programs, transforming the pre and postconditions by swapping the memories they refer to.
Examples:

$$\frac{\{P^{-1}\}\ c_2 \sim c_1\ \{Q^{-1}\}}{\{P\}\ c_1 \sim c_2\ \{Q\}}\ (\text{pRHL})\quad \text{symmetry}$$

◎ transitivity

Syntax: transitivity c ($P_1 \Longrightarrow Q_1$) ($P_2 \Longrightarrow Q_2$). In pRHL, applies the transitivity of program equivalence using the specified program and specifications. When the goal is a judgment on procedures, c should be a procedure. When the goal is a judgment on statements, c should be a statement, and the tactic then takes a

side argument, used to decide the procedure context under which local variables from c are evaluated.
Examples:

$$\frac{\begin{array}{c}\forall m_1\ m_2.\, m_1\ P\ m_2 \Rightarrow \exists m.\, m_1\ P_1\ m \wedge m\ P_2\ m_2 \\ \forall m_1\ m\ m_2.\, m_1\ Q_1\ m \Rightarrow m\ Q_2\ m_2 \Rightarrow m_1\ Q\ m_2 \\ \{P_1\}\ f_1 \sim f\ \{Q_1\} \qquad \{P_2\}\ f \sim f_2\ \{Q_2\}\end{array}}{\{P\}\ f_1 \sim f_2\ \{Q\}}\ (\textsc{prhl})\quad \textsf{transitivity f } (P_1 \Longrightarrow Q_1)\ (P_2 \Longrightarrow Q_2)$$

$$\frac{\begin{array}{c}\forall m_1\ m_2.\, m_1\ P\ m_2 \Rightarrow \exists m.\, m_1\ P_1\ m \wedge m\ P_2\ m_2 \\ \forall m_1\ m\ m_2.\, m_1\ Q_1\ m \Rightarrow m\ Q_2\ m_2 \Rightarrow m_1\ Q\ m_2 \\ \{P_1\}\ s_1 \sim s\ \{Q_1\} \qquad \{P_2\}\ s \sim s_2\ \{Q_2\}\end{array}}{\{P\}\ s_1 \sim s_2\ \{Q\}}\ (\textsc{prhl})\quad \textsf{transitivity}\{1\}\ \{\ \textsf{s}\ \}\ (P_1 \Longrightarrow Q_1)\ (P_2 \Longrightarrow Q_2)$$

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

◎ conseq

Syntax: conseq <spec>. Rule of consequence. Proves a specification by weakening of a stronger result. Any one of the specification places can be filled with a wildcard _ to keep the value it contains in the current goal and trivially discharge the corresponding subgoal.
Examples:

$$\frac{P' \Rightarrow P \qquad Q \Rightarrow Q' \qquad \{P\}\ c \sim c'\ \{Q\}}{\{P'\}\ c \sim c'\ \{Q'\}}\ (\textsc{prhl})\quad \textsf{conseq } (\_\colon \textsf{P} \Longrightarrow \textsf{Q})$$

$$\frac{Q \Rightarrow Q' \qquad \{P'\}\ c \sim c'\ \{Q\}}{\{P'\}\ c \sim c'\ \{Q'\}}\ (\textsc{prhl})\quad \textsf{conseq } (\_\colon \_ \Longrightarrow \textsf{Q})$$

$$\frac{P' \Rightarrow \delta \diamond \delta' \qquad P' \Rightarrow P \qquad Q \diamond^{\uparrow} Q' \qquad \{P\}\ c\ \{Q\} \diamond \delta}{\{P'\}\ c\ \{Q'\} \diamond \delta'}\ (\textsc{phl})\quad \textsf{conseq } (\_\colon \textsf{P} \Longrightarrow \textsf{Q}\colon \delta)$$

$$\frac{P' \Rightarrow P \qquad Q \Rightarrow Q' \qquad \{P\}\ c\ \{Q\}}{\{P'\}\ c\ \{Q'\}}\ (\textsc{hl})\quad \textsf{conseq } (\_\colon \textsf{P} \Longrightarrow \textsf{Q})$$

Note: The pHL variant can also be used to strenghten the relation $\diamond$ into an equality by forcing the equality into the specification. For example, the following is a valid application of conseq.

$$\frac{P' \Rightarrow \delta = \delta' \qquad P' \Rightarrow P \qquad Q \Leftrightarrow Q' \qquad \{P\}\ c\ \{Q\} = \delta}{\{P'\}\ c\ \{Q'\} \leq \delta'}\ (\textsc{phl})\quad \textsf{conseq } (\_\colon \textsf{P} \Longrightarrow \textsf{Q}\colon = \delta)$$

Syntax: conseq <**lemma**>. Only applies to judgments on procedures. Same as conseq <spec>, but the specification to use is inferred from the lemma provided.

Raises an error if the lemma does not refer to the expected procedure(s). All variants of conseq may take lemmas in place of explicit specifications with the same effect, in which case they must be applied to judgments on procedures.

Syntax: conseq* <spec>. Same as conseq, but the subgoal corresponding to the postcondition is refined by a "may modify" analysis. All variants of conseq can be refined using the $*$, with the same effect.

Syntax: conseq <prhl> <hl> <hl>. Combine relational and non-relational specifications to prove a relational specification. Either one of the Hoare logic specifications can be replaced with a wildcard.
Examples:

$$\frac{P' \Rightarrow P \wedge P_1\langle 1\rangle \wedge P_2\langle 2\rangle \qquad Q \wedge Q_1\langle 1\rangle \wedge Q_2\langle 2\rangle \Rightarrow Q' \qquad \{P_1\}\ c_1\ \{Q_1\} \qquad \{P_2\}\ c_2\ \{Q_2\} \qquad \{P\}\ c_1 \sim c_2\ \{Q\}}{\{P'\}\ c_1 \sim c_2\ \{Q'\}}\ (\text{pRHL})$$ conseq (_: P ⟹ Q) (_: P₁ ⟹ Q₁) (_: P₂ =

Syntax: conseq <prhl> <phl> | conseq <prhl> _ <phl>. Strengthen a relational specification where one of the programs is empty into a non-relational specification about the non-empty program. Fails if the program expected to be empty is not. For uniformity and simplicity of use, this variant also allows the user to strengthen the pRHL judgment before abstracting its relational aspects. The general rule below can be understood more clearly when $P = P'$ and $Q = Q'$.
Examples:

$$\frac{P' \Rightarrow P \qquad Q \Rightarrow Q' \qquad P \Rightarrow P_1 \qquad Q_1 \Rightarrow Q \qquad \{P_1\}\ c\ \{Q_1\} = 1}{\{P'\}\ c \sim \mathsf{skip}\ \{Q'\}}\ (\text{pRHL})$$ conseq (_: P ⟹ Q) (_: P₁ =

$$\frac{P \Rightarrow P_2 \qquad Q_2 \Rightarrow Q \qquad \{P_2\}\ c\ \{Q_2\} = 1}{\{P\}\ \mathsf{skip} \sim c\ \{Q\}}\ (\text{pRHL})$$ conseq (_: _ ⟹ _) _ (_: P₂ ⟹ Q₂: =1%r)

FiXme Note: Missing descriptions of any variants of conseq?

◎ case

Syntax: case e. Split the current pRHL, pHL or HL goal by doing a case analysis in the precondition.
Example:

$$\frac{\{P \wedge C\}\ c_1 \sim c_2\ \{Q\} \qquad \{P \wedge \neg C\}\ c_1 \sim c_2\ \{Q\}}{\{P\}\ c_1 \sim c_2\ \{Q\}}\ (\text{pRHL})$$ case C

$$\frac{\{P \wedge C\}\ c\ \{Q\} \diamond \delta \qquad \{P \wedge \neg C\}\ c\ \{Q\} \diamond \delta}{\{P\}\ c\ \{Q\} \diamond \delta}\ (\text{pHL})$$ case C

$$\frac{\{P \wedge C\}\ c\ \{Q\} \qquad \{P \wedge \neg C\}\ c\ \{Q\}}{\{P\}\ c\ \{Q\}}\ (\text{HL})$$ case C

◎ phoare split

Syntax: phoare split$\delta_A\delta_B\delta_{AB}$. Splits a pHL judgment whose postcondition is a conjunction or disjunction into three pHL judgments following the definition of the probability of a disjunction of events.
Examples:

$$\frac{\delta_A + \delta_B - \delta_{AB} \diamond \delta \qquad \{P\}\ c\ \{A\} \diamond \delta_A \qquad \{P\}\ c\ \{B\} \diamond \delta_B \qquad \{P\}\ c\ \{A \wedge B\} \diamond^{-1} \delta_{AB}}{\{P\}\ c\ \{A \vee B\} \diamond \delta} \quad (\text{pHL}) \quad \text{phoare split } \delta_A\ \delta_B\ \delta_{AB}$$

$$\frac{\delta_A + \delta_B - \delta_{AB} \diamond \delta \qquad \{P\}\ c\ \{A\} \diamond \delta_A \qquad \{P\}\ c\ \{B\} \diamond \delta_B \qquad \{P\}\ c\ \{A \vee B\} \diamond^{-1} \delta_{AB}}{\{P\}\ c\ \{A \wedge B\} \diamond \delta} \quad (\text{pHL}) \quad \text{phoare split } \delta_A\ \delta_B\ \delta_{AB}$$

Syntax: phoare split ! $\delta_\top$ $\delta_!$. Splits a pHL judgment into two judgments whose postcondition are true and the negation of the original postcondition, respectively.
Examples:

$$\frac{\delta_\top - \delta_! \diamond \delta \qquad \{P\}\ c\ \{\mathsf{true}\} \diamond \delta_\top \qquad \{P\}\ c\ \{!Q\} \diamond^{-1} \delta_!}{\{P\}\ c\ \{Q\} \diamond \delta} \quad (\text{pHL}) \quad \text{phoare split ! } \delta_\top\ \delta_!$$

Syntax: phoare split$\delta_A\delta_{!A}$: A. Splits a pHL judgment following an event A.
Examples:

$$\frac{\delta_A + \delta_{!A} \diamond \delta \qquad \{P\}\ c\ \{Q \wedge A\} \diamond \delta_A \qquad \{P\}\ c\ \{Q \wedge \neg A\} \diamond \delta_{!A}}{\{P\}\ c\ \{Q\} \diamond \delta} \quad (\text{pHL}) \quad \text{phoare split } \delta_A\ \delta_{!A}\text{: A}$$

◎ byequiv

Syntax: byequiv [option]? <spec>. Derives a probability relation from a pRHL judgement on the procedures involved. <spec> can include wildcards when the tactic should infer the pre or postcondition. In addition, <spec> can be extended with a failure event to infer precise applications of the Fundamental Lemma.
Options: By default, (eq option) specification inference attempts to infer a conjunction of equalities sufficient to imply the desired relation. Passing the −eq option overrides this behaviour, instead using the trivial relation on events.

Examples:

$$\frac{\{P\}\ f_1 \sim f_2\ \{Q\} \qquad m_1[\mathsf{arg} \mapsto \vec{a}_1]\ P\ m_2[\mathsf{arg} \mapsto \vec{a}_2] \qquad Q \Rightarrow E_1\{1\} \Leftrightarrow E_2\{2\}}{\mathbf{Pr}[m_1, f_1(\vec{a}_1) : E_1] = \mathbf{Pr}[m_2, f_2(\vec{a}_2) : E_2]}$$ byequiv (_: P ⟹ Q)

$$\frac{\{P\}\ f_1 \sim f_2\ \{Q\} \qquad m_1[\mathsf{arg} \mapsto \vec{a}_1]\ P\ m_2[\mathsf{arg} \mapsto \vec{a}_2] \qquad Q \Rightarrow E_1\{1\} \Rightarrow E_2\{2\}}{\mathbf{Pr}[m_1, f_1(\vec{a}_1) : E_1] \leq \mathbf{Pr}[m_2, f_2(\vec{a}_2) : E_2]}$$ byequiv (_: P ⟹ Q)

$$\frac{\{P\}\ f_1 \sim f_2\ \{Q\} \qquad m_1[\mathsf{arg} \mapsto \vec{a}_1]\ P\ m_2[\mathsf{arg} \mapsto \vec{a}_2] \qquad Q \Rightarrow E_2\{2\} \Rightarrow E_1\{1\}}{\mathbf{Pr}[m_1, f_1(\vec{a}_1) : E_1] \geq \mathbf{Pr}[m_2, f_2(\vec{a}_2) : E_2]}$$ byequiv (_: P ⟹ Q)

$$\frac{\begin{array}{c}\{P\}\ f_1 \sim f_2\ \{Q\} \\ m_1[\mathsf{arg} \mapsto \vec{a}_1]\ P\ m_2[\mathsf{arg} \mapsto \vec{a}_2] \qquad Q \Rightarrow \neg B_2\{2\} \Rightarrow E_1\{1\} \Rightarrow E_2\{2\}\end{array}}{\mathbf{Pr}[m_1, f_1(\vec{a}_1) : E_1] \leq \mathbf{Pr}[m_2, f_2(\vec{a}_2) : E_2] + \mathbf{Pr}[m_2, f_2(\vec{a}_2) : B_2]}$$ byequiv (_: P ⟹ Q)

$$\frac{\begin{array}{c}\{P\}\ f_1 \sim f_2\ \{Q\} \qquad m_1[\mathsf{arg} \mapsto \vec{a}_1]\ P\ m_2[\mathsf{arg} \mapsto \vec{a}_2] \\ Q \Rightarrow (B_1\{1\} \Leftrightarrow B_2\{2\}) \wedge (\neg B_2\{2\} \Rightarrow E_1\{1\} \Leftrightarrow E_2\{2\})\end{array}}{|\mathbf{Pr}[m_1, f_1(\vec{a}_1) : E_1] - \mathbf{Pr}[m_2, f_2(\vec{a}_2) : E_2]| \leq \mathbf{Pr}[m_2, f_2(\vec{a}_2) : B_2]}$$ byequiv (_: P ⟹ Q) : B_1

$$\frac{\{P\}\ f_1 \sim f_2\ \{E_1\{1\} \Leftrightarrow E_2\{2\}\} \qquad m_1[\mathsf{arg} \mapsto \vec{a}_1]\ P\ m_2[\mathsf{arg} \mapsto \vec{a}_2]}{\mathbf{Pr}[m_1, f_1(\vec{a}_1) : E_1] = \mathbf{Pr}[m_2, f_2(\vec{a}_2) : E_2]}$$ byequiv [−eq] (_: P ⟹ _)

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

◎ byphoare

Syntax: byphoare [option]? <spec>. Derives a probability relation from a pHL judgement on the procedure involved. <spec> can include wildcards when the tactic should infer the pre or postcondition.
Options: By default, (eq option) specification inference attempts to infer a conjunction of equalities sufficient to imply the desired relation. Passing the −eq option overrides this behaviour, instead using the trivial relation on events.
Examples:

$$\frac{\{P\}\ f\ \{Q\} = \delta \qquad P\ m[\mathsf{arg} \mapsto \vec{a}] \qquad \forall m'.\, Q\ m' \Leftrightarrow E\ m'}{\mathbf{Pr}[m, f(\vec{a}) : E] = \delta}$$ byphoare (_: P ⟹ Q)

Syntax: byphoare <**lemma**>. Same as byphoare <spec>, but the specification to use is inferred from the lemma provided. Raises an error if the lemma does not refer to the expected procedure. Inference options have no effect in this setting.

◎ hoare

Syntax: hoare <spec>. Derives a null probability from a HL judgement on the procedure involved. hoare can also be used to derive pHL judgments and certain probability inequalities by automatically applying conseq (p. 8).

Examples:

$$\frac{\{\mathsf{true}\}\ f\ \{\neg Q\}}{\mathbf{Pr}[m, f(\vec{a}) : Q] = 0}\ \text{hoare}$$

$$\frac{\{\mathsf{P}\}\ f\ \{\neg Q\}}{\{P\}\ f\ \{Q\} \leq 0}\ \text{hoare}$$

◎ bypr

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

$$\frac{\begin{array}{c}\forall m_1, m_2, a.\, r_1 = a \Rightarrow r_2 = a \Rightarrow m_1\ Q\ m_2 \\ \forall \vec{a}_1, \vec{a}_2, m_1, m_2, a.\, m_1[\mathsf{arg} \mapsto \vec{a}_1]\ P\ m_2[\mathsf{arg} \mapsto \vec{a}_2] \Rightarrow \mathbf{Pr}[m_1, f_1(\vec{a}_1) : a = r_1] = \mathbf{Pr}[m_2, f_2(\vec{a}_2) : a = r_2]\end{array}}{\{P\}\ f_1 \sim f_2\ \{Q\}}\ (\text{pRH}$$

$$\frac{\forall m, \vec{a}.\, P\ m[\mathsf{arg} \mapsto \vec{a}] \Rightarrow \mathbf{Pr}[m, f(\vec{a}) : E] \diamond \delta}{\{P\}\ f\ \{E\} \diamond \delta}\ (\text{pHL})\ \text{bypr}$$

$$\frac{\forall m, \vec{a}.\, P\ m[\mathsf{arg} \mapsto \vec{a}] \Rightarrow \mathbf{Pr}[m, f(\vec{a}) : \neg E] = 0\%r}{\{P\}\ f\ \{E\}}\ (\text{HL})\ \text{bypr}$$

◎ exfalso

Syntax: exfalso. Combines conseq (p. 8), byequiv (p. 10), byphoare (p. 11), hoare (p. 11) and bypr (p. 12) to strengthen the precondition into false and to discharge the resulting trivial goal.
Examples:

$$\frac{P \Rightarrow \mathsf{false}}{\{P\}\ c \sim c'\ \{Q\}}\ (\text{pRHL})\ \text{exfalso}$$

$$\frac{P \Rightarrow \mathsf{false}}{\{P\}\ c\ \{Q\} \diamond \delta}\ (\text{pHL})\ \text{exfalso}$$

$$\frac{P \Rightarrow \mathsf{false}}{\{P\}\ c\ \{Q\}}\ (\text{HL})\ \text{exfalso}$$

FiXme Note: Move exfalso to automatic tactics?

Reasoning on Programs Unless specified, the following program logic tactics operate on a program's last instruction. Although we describe these tactics as if they operated on single instructions, their practical implementation automatically and implicitly applies tactic seq (p. 13) to deal with context when necessary.

For simple proofs, it is often enough to simply apply the program tactic corresponding to the last instruction in the program and let smt deal with the verification condition once the program has been exhausted.

Most of the program reasoning tactics discussed in this paragraph have two modes when used on pRHL proof obligations. Their default mode is to operate on both programs at once. When a side is specified (using <tactic>{1} or <tactic>{2}), a one-sided variant is used. Apart from the **if** (p. 15) tactic, the one-sided variant is in fact a combination of the pHL tactic and conseq (p. 8).

◎ skip

Syntax: skip. Hoare logic rules for empty statement.
Examples:

$$\frac{P \Rightarrow Q}{\{P\}\ \mathsf{skip} \sim \mathsf{skip}\ \{Q\}}\quad (\text{pRHL})\quad \text{skip}$$

$$\frac{P \Rightarrow Q}{\{P'\}\ \mathsf{skip}\ \{Q'\} \diamond 1}\quad (\text{pHL})\quad \text{skip}$$

$$\frac{P \Rightarrow Q}{\{P\}\ \mathsf{skip}\ \{Q\}}\quad (\text{HL})\quad \text{skip}$$

Note: Note that the pHL rule forces the bound of the goal to be 1. If you end up with an empty program and a bound other than 1, you might want to use hoare (p. 11) or conseq (p. 8). If neither of these work, you should probably have used seq (p. 13) or phoare split (p. 9) earlier on in your proof.

◎ seq

Syntax: seq n_1 n_2: R. Relational sequence rule.
Examples:

$$\frac{\{P\}\ c_1 \sim c_2\ \{R\} \qquad \{R\}\ c_1' \sim c_2'\ \{Q\}}{\{P\}\ c_1; c_1' \sim c_2; c_2'\ \{Q\}}\quad (\text{pRHL})\quad \text{seq } |c_1|\ |c_2|\text{: R}$$

Syntax: seq n: R. Non-relational possibilistic sequence rule.
Examples:

$$\frac{\{P\}\ c\ \{R\} \qquad \{R\}\ c'\ \{Q\}}{\{P\}\ c; c'\ \{Q\}}\quad (\text{HL})\quad \text{seq } |c|\text{: R}$$

Syntax: seq n: R δ_1 δ_2 δ_3 δ_4 I. Non-relational probabilistic sequence rule. Argument I is optional (and defaults to true). When one of (δ_1, δ_2) (resp. (δ_3, δ_4)) is 0, the other can be replaced with a wildcard _, and the corresponding goal is not generated, as it is not relevant to the proof. When none of the δs are given, the following default values are used: $\delta_1 = 1$, $\delta_2 = \delta$, $\delta_3 = 0$.

Examples:

$$\frac{\begin{array}{ccc}\{P\}\ c\ \{I\} & \{P\}\ c\ \{R\} \diamond \delta_1 & \{R \wedge I\}\ c'\ \{Q\} \diamond \delta_2 \\ \{P\}\ c\ \{\neg R\} \diamond \delta_3 & \{\neg R \wedge I\}\ c'\ \{Q\} \diamond \delta_4 & \delta_1\delta_2 + \delta_3\delta_4 \diamond \delta\end{array}}{\{P\}\ c; c'\ \{Q\} \diamond \delta}\ (\text{PHL})\quad \mathsf{seq}\ |c|\colon \mathsf{R}\ \delta_1\ \delta_2\ \delta_3\ \delta_4\ \mathsf{I}$$

$$\frac{\begin{array}{ccccc}\{P\}\ c\ \{\mathsf{true}\} & & & & \\ \{P\}\ c\ \{R\} \diamond \delta_1 & \{R \wedge I\}\ c'\ \{Q\} \diamond \delta_2 & \{\neg R \wedge I\}\ c'\ \{Q\} \diamond 0 & & \delta_1\delta_2 \diamond \delta\end{array}}{\{P\}\ c; c'\ \{Q\} \diamond \delta}\ (\text{PHL})\quad \mathsf{seq}\ |c|\colon \mathsf{R}\ \delta_1\ \delta_2\ \_0$$

Note: Since most tactics implicitly apply the seq (p. 13) rule, most PHL tactics take optional final arguments corresponding to the δs and I.

◎ sp

Syntax: sp. Computes the strongest postcondition of a straightline deterministic prefix of the program(s) that is implied by the current precondition. sp also consumes deterministic **if** statements (when both branches are deterministic straightline code without procedure calls).

Syntax: sp n_1 n_2. In PRHL, let sp consume *exactly* n_1 statements of the left program and n_2 statements of the right program.

Syntax: sp n. In PHL and HL, let sp consume *exactly* n statements of the program.

◎ wp

Syntax: wp. Computes the weakest precondition of a straightline deterministic suffix of the program(s) that implies the current postcondition. wp also consumes deterministic **if** statements (when both branches are deterministic straightline code without procedure calls).

Syntax: wp n_1 n_2. In PRHL, let wp consume *exactly* n_1 statements of the left program and n_2 statements of the right program.

Syntax: wp n. In PHL and HL, let wp consume *exactly* n statements of the program.

◎ rnd

Syntax: rnd | rnd f | rnd f f^{-1}. In PRHL, compute the probabilistic weakest precondition of two random samplings whose results are equally distributed, by exhibiting a probability-preserving bijection between the two distributions. When f^{-1} is f, it can be omitted. When f is the identity, both f and f^{-1} can be omitted.

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.

Syntax: rnd. In HL, compute the possibilistic weakest precondition of a random sampling operation.

◎ **if**

Syntax: **if**. Operates on the *first* statement (instead of the last one). If the first statement in the program is **if** (b) $\{c_1\}$ **else** $\{c_2\}$, applying tactic **if** is equivalent to case b; [rcondt 1⇒ //= | rcondf 1⇒ //=] (see case (p. 9), rcondt (p. 16), rcondf (p. 16)).

◎ **while**

Syntax: **while** I. In pRHL and HL, as well as upper-bounding pHL judgments, performs a weakest precondition computation over a loop using I as invariant. This generates two subgoals: one explaining that I is a valid loop invariant, and the other explaining that the invariant is initially true and that it is sufficient to establish the current postcondition.

Syntax: **while** I v. Where v is an integer-valued expression. In pHL, performs a weakest precondition computation over a loop, using I as invariant and v as a decreasing variant to prove termination. In addition to the two invariant-related subgoals (see above), two subgoals regarding the variant are generated; one requiring that the variant be less than 0 exactly when the loop condition is false, and the other requiring that the variant decreases strictly.

Note: More complex variants of the **while** tactic exist, useful in particular for dealing in pHL with loops whose termination is not variant-based. However, these advanced variants are currently undocumented.

◎ call

All variants of the call tactic implicitly make use of a frame rule, based on a "may modify" analysis.
Syntax: call (_: P ⟹ Q). Compute the precondition of a procedure call using the given specification for the procedure. As a side-goal, prove that the procedure fulfills the given specification.
As with other tactics, the specification (_: P ⟹ Q) can be replaced with a lemma from which the specification is inferred.

Syntax: call (_: I). Uses invariant I to infer a specification for use with tactic.
In pRHL, and if A is the abstract module, call (_: I) is equivalent to call (_: ={arg, $globA$} ∧ I ⟹ ={res, $globA$} ∧
In pHL and HL, call (_: I) is equivalent to call (_: I ⟹ I); first **proc** I.

Syntax: call (_: B, I). On pRHL abstract procedures only. If A is the abstract module, call (_: B, I) is equivalent to call (_:¬B ∧ ={arg, $globA$} ∧ I ⟹¬B ⇒ ={res, $globA$} ∧ I); first **proc** B I.

Syntax: call (_: B, I, I'). On pRHL abstract procedures only. If A is the abstract module, call (_: B, I) is equivalent to call (_:¬B ∧ ={arg, $globA$} ∧ I ⟹ **if**¬B then ={res, $globA$} ∧ I **else** I'))

Note: When using the invariant-based variants of call, error messages may be originating from the underlying application of **proc** (p. 15). In particular, when using them to deal with abstract procedure calls, the invariant *should not* refer to memory locations the abstract procedure may modify.

◎ **proc**

Syntax: **proc**. Derive a specification for a *concrete* procedure from a specification on its code.

Syntax: **proc** I. Derive a specification for an *abstract* procedure from an invariant on the oracles it may query.

Syntax: **proc** B I. Derive a specification for an *abstract* procedure from an "upto-failure" invariant on the oracles it may query. The failure event B is evaluated in the right memory. The left oracles must be lossless once the bad event occurs. The right oracles must guarantee the stability of the failure event with probability 1.

Syntax: **proc** B I I'. Similar to **proc** B I, with an additional invariant once the bad event occurs. This is particularly useful when additional facts about the state need to be known to prove the losslessness and stability conditions.

Syntax: **proc***. Derive a specification on procedures from a specification on a program whose code consists in a call to that procedure. This tactic is particularly useful in combination with inline (p. 16) when faced with a pRHL judgment where one of the procedures is concrete and the other is abstract.

Transforming Programs

◎ swap

All versions of the tactic work for pRHL (an optional side can be given), pHL and HL.
Syntax: swap p_1 p_2 p_3. Swaps the code between positions p_1 and p_2 with the code between positions p_2 and p_3. That is, assuming that c_1 and c_2 are syntactically independent, that c_1 is between positions p_1 and p_2 and that c_2 is between positions p_2 and p_3, the tactic implements the following rule:

$$\frac{\{P\}\ c; c_2; c_1; c_3\ \{Q\}}{\{P\}\ c; c_1; c_2; c_3\ \{Q\}}[\quad p_1\ p_2\ p_3] \qquad \text{swap}$$

Syntax: swap k. If k is positive (negative) then [swap k] moves the first (last) instruction k positions forwards (backwards).

Syntax: swap p k. Moves the p^{th} instruction forwards or backwards.

Syntax: swap [p_1..p_2] k. Moves the instructions between positions p_1 and p_2 forwards or backwards.

◎ inline

FiXme Note: Missing description of inline.

◎ rcondf

FiXme Note: Missing description of rcondf.

◎ rcondt

FiXme Note: Missing description of rcondt.

◎ splitwhile

FiXme Note: Missing description of splitwhile.

◎ unroll

FiXme Note: Missing description of unroll.

◎ fission

FiXme Note: Missing description of fission.

◎ fusion

FiXme Note: Missing description of fusion.

◎ alias

FiXme Note: Missing description of alias.

◎ cfold

FiXme Note: Missing description of cfold.

◎ kill

FiXme Note: Missing description of kill.

◎ modpath

FiXme Note: Missing description of modpath.

Automated Tactics

◎ auto

FiXme Note: Missing description of auto.

◎ sim

Syntax: sim <pos>? <hintgeqs>∗ <hintinv>? <eqs>?.

where

<pos>	=	<uint> <uint>
<hintgeqs>	=	(<procname>? ∼ <procname>? : <formula>)
	|	(_? : <formula>
<eqs>	=	: <formula>

FiXme Note: Missing description of sim.

Advanced Tactics

◎ eager

FiXme Note: Missing description of eager. FiXme Note: Missing descriptions for all eager <tactic> variants.

◎ fel

FiXme Note: Missing description of fel.

Index of Tactics *and al*

List of Corrections