PREMIER MINISTRE
Liberté
Égalité
Fraternité

Requirements on the Use of Rocq in the Context of Common Criteria Evaluations

French National Cybersecurity Agency (ANSSI)
INRIA

v1.2 14/04/2025

Versions	Date	Modifications
1.2	14/04/2025	Updating the document to take into account the renaming from “The Coq Proof Assistant” to “The Rocq Prover” as of the release of 9.0.
1.1	09/12/2021	Fixing some typographical errors and adding developers rules 4.14, 4.15 and 4.16.
1.0	23/09/2020	Creation

Contents

Chapter 1

Introduction

The Rocq prover[1] is a formal system which provides both a pure, functional programming language with dependent types, and an environment to write machine-checked proofs. Using Rocq, it is possible to model a system, to formalize properties this system is expected to satisfy, and to prove the correctness of the model with respect to these properties. Rocq has already been leveraged for Common Criteria evaluations with formal developments in the context of the French scheme. The present document assumes readers are already familiar with Rocq[2], and introduces requirements to write formal developments suitable to take part in CC evaluations. Evaluators are expected to assert that the developers' requirements have been satisfied during the evaluation process. In addition, this document also introduces requirements specifically for evaluators.

This document proceeds as follows. First, we give a quick introduction on the Rocq prover architecture, to distinguish between the system kernel —which has to be trusted— and the system "extensions" —which should be used only in ways that cannot endanger the system consistency (Chapter 2). Then, we detail how the Rocq distribution (*i.e.*, Rocq, and its standard library) shall be configured in order for its kernel to remain consistent (Chapter 3). Lastly, we provide development requirements which focus on writing readable and reviewable formal developments using Rocq (Chapter 4).

[1] Formerly known as the Coq Proof Assistant. See `https://rocq-prover.org/doc/V9.0.0/refman/changes.html#summary-of-changes` for more details.

[2] The reader willing to learn Rocq can refer to more appropriate resources, such as *The Software Foundations* series by Benjamin C. Pierce *et al.* or the *Coq'Art* by Yves Bertot and Pierre Castéran.

Chapter 2

Rocq Architecture

The central idea of the Rocq system is that the correctness of proofs is guaranteed by a *type-checking* algorithm. A corollary of this central idea is that proof search and proof verification are two separate tasks, where only proof verification is crucial for trustworthiness. When a proof search procedure produces a result, this result is transferred to the proof verification algorithm which checks it for consistency. Following this principle, an error in a clever proof search procedure should not endanger the validity of its results, because the latter are going to be checked again by the type-checking algorithm.

As a consequence, the type-checking algorithm is placed at the heart of all trust questions, and is therefore often referred to as the *kernel* of Rocq. However, the trustworthiness of Rocq cannot be reduced to the trustworthiness of its kernel. We should also add considerations about the data paths between the human user and the type-checking algorithm. When a piece of data is displayed by the Rocq system, what part of this data was actually checked by the kernel? When the user submits a question to the proof system, which part of the question is submitted to the kernel? This section attempts to clarify these questions.

We must distinguish between 5 kinds of data: (1) vernacular commands, (2) programs and formulas, (3) proofs steps, also sometimes referred to as *tactics*, (4) Rocq extensions given in the form of OCaml sources, compiled as plugins and loaded into the Rocq program, and (5) precompiled libraries. Except for precompiled libraries, these data are inputs submitted by human users in text form.

Vernacular commands. Rocq provides many vernacular commands for various purposes, nature, and criticality. Some commands modify the way data will be understood (*e.g.*, adding parsing directives, defining new tactics, storing theorems into so-called hint databases used in proofs search automation procedures…). These commands do not interact with the type-checker. Other commands really imply verification by the type-checker. Usually, these directives are related to adding a new definition or a new theorem in Rocq's internal database. At this moment, the new definition or theorem is verified by the kernel. For a theorem in particular, it means that the whole proof of the theorem is verified for consistency, including the adequation with the claimed statement. Finally, several commands can be used to query the current state of the kernel, without modifying it, such as `About`, or `Print`. These queries are usually not recorded in Rocq source

files, simply because their presence is not needed. However, they can be useful to help better understanding the contents of a formal development.

Programs and formulas. Text representing programs or formulas is written in a language called GALLINA, and is usually part of a vernacular command whose purpose is to add a new definition to the Rocq kernel (*e.g.*, `Definition`, `Lemma`, etc.). Most of the time, the text is not complete to the level accepted by the kernel, so a phase of *elaboration* is performed, to fill in some blanks that would be too cumbersome to require from users. Once this elaboration is performed, the program or formula is usually passed through the kernel before exploitation by the encapsulating command.

The degree of transformation performed by the elaboration phase is really dependent on the vernacular command used by developers. The Russel framework (enabled by the `Program` command, or the `program` attribute in recent Rocq versions) and the `Function` command are two good examples of heavy elaboration phases. Since the kernel verifies the correctness of a term with respect to a type, it is important to verify the exact type of newly introduced definitions (for instance with `About`), to ensure they correspond to the developers' intents.

Proof steps. The third category of input from users consists in the text of goal-directed proofs. This is usually organized as a semi-structured text, containing steps that transform an initial question (a theorem statement) into simpler questions (sub-goals), until all goals have been solved. In a Rocq source file, users only record the steps that are sent to the Rocq system to modify the goal state, but this script can usually only be understood fully by observing in lock step the commands sent by the user (and recorded in the Rocq source) and the answers given at each step by the proof system. From a trust point of view, the steps taken by each tactic may or may not involve verification by the kernel. It is not crucial, because the operations performed at each step are gathered in a *proof term* that is verified by the type-checker at the end of the proof of each given theorem, when the command `Qed` or `Defined` is called. The artifact "remembered" by Rocq is this proof term, not the proof scripts used to generate it.

Rocq plugins. The fourth category of input from users consists of extensions to the Rocq system that can be imported in the form of *plugins*. This category of inputs is usually chosen by expert users who wish to add powerful proof search capabilities to the system. Most of the time, the objective is to add new tactics that can be used for the third category of inputs. A few of the plugins are provided in the Rocq sources: `btauto`, `cc`, `firstorder`, `fourier`, `micromega`, `nsatz`, `omega`, `setoid_ring`, `ssr` and `ssrmatching` are mostly concerned with adding new tactics; `funind` is concerned with generating new proofs automatically (and these proofs, of course, go through verification by the type-checker), `syntax` is concerned with providing comfortable notations for numbers, and `extraction` is concerned with producing code for outside use.

From a trust point of view, code written in OCaml is sensitive. The various APIs provided by Rocq at the level of OCaml are not protective enough to avoid a malicious code writer modifies data after it has been verified. In other words, it is possible for a Rocq plugin implemented in OCaml to trick the kernel into accepting ill-formed terms. For this reason, only trusted plugins should be used in developments where trust is important.

Precompiled libraries. The fifth category of data consumed by the Rocq system is made of precompiled libraries. Every time users call the `rocq compile` or `rocq c` command, a precompiled file (with the suffix `.vo`) is produced from a source file (with the suffix `.v`) file. In its current way of operation, the Rocq system produces `.vo` files by applying its full verification criteria. In other words, the type-checking algorithm validates the content of a `.vo` file before it is produced. When loading a `.vo` file is requested by a user (usually by calling the `Require` command), the `rocq c` and `rocq top` commands trust the content of this file blindly. One could envision a scenario where `.vo` files are tampered with between production by Rocq and consumption by Rocq in return, in such a way that logical consistency could be compromised.

The Rocq system provides a tool called `rocqchk` that verifies the `.vo` files. Contrary to the process described in the previous paragraph, this tool reloads all pre-existing libraries and *verifies* them using the type-checker. Besides, `rocqchk` does not load any plugins to verify its inputs, and therefore shall uncover inconsistencies introduced by incorrect plugins. For trust purposes, this tool is important.

Chapter 3

Rocq Standard Distribution

In Section 2, we have described the architecture of Rocq, and we have identified its kernel. In this section, we provide recommendations whose purpose is to correctly configure and use this kernel in a trustworthy manner.

3.1 Version Accountability

Rocq is actively developed by a large team of contributors. A new version is released every six months, and (significant) features are regularly introduced (*e.g.*, cumulative inductive types in 8.7, mutually recursive records in 8.9, `SProp` and primitive types in 8.10). In addition to these new features, the Rocq development team is engaged in a significant effort to improve the overall quality of the Rocq code base. The Rocq development team has significantly increased their infrastructure and continuous integration efforts in order to provide a reliable tool with a strong foundation and good development practices.

Developers Rule 3.1. When starting a new Rocq project, developers shall always use the latest release of Rocq.

The Rocq development team maintains a (partial) list of critical bugs, which is available in the Rocq repository. It is their commitment that this list becomes, over time, a reliable source for Rocq versions accountability.

Developers Rule 3.2. Developers shall clearly identify which version of Rocq they have used.

Evaluators Rule 3.1. Evaluators shall verify that the version of Rocq used in the evaluated formal development is either not subject to known critical bugs or that the formal development does not use features impacted by said critical bugs.

Rocq is a mature theorem prover, and comes with a significant number of third-party libraries.

Developers Rule 3.3. Developers shall clearly identify which third-party libraries they use, and shall provide them as part of their formal development project.

The French Certification Body keeps records of Rocq third-party libraries which can be used within the French scheme, optionally with certain usage restrictions.

Evaluators Rule 3.2. For third-party libraries unknown to the French Certification Body, Evaluators shall review them with respect to the requirements of this document, with the exceptions of Section 4.3.

This additional evaluation task is based on one of the following approaches:

- Evaluators review the subsets of third-party libraries which are effectively used in the formal development. As a consequence, the result of this analysis may not be reusable in the context of future evaluations.
- Evaluators review the third-party libraries in their entirety. As a consequence, the result of this analysis is reusable in the context of future evaluations.

3.2 rocqchk

Although the Rocq kernel is advertised as small, Rocq as a formal system comes with a lot of extensions whose degree of stability and maturity may vary.

By means of plugins, Rocq can gain convenient new functionalities, but it can be at the cost of trust. Most plugins will provide new ways to construct Rocq terms during the compilation process. These terms are then stored in `.vo` files. `rocqchk` is a tool distributed with Rocq, whose purpose is to provide a small checker (small by comparison with `rocq c`) for those files.

Evaluators Rule 3.3. Evaluators shall use `rocqchk` to recursively verify Rocq build artifacts (`.vo`), with the following criteria:

- The output of `rocqchk` shall be examined to assert the main theorems have effectively been checked.
- The `-norec` command-line argument of `rocqchk` shall not be used
- If `rocqchk` finds an inconsistency, evaluators shall consider the work units related to the affected model as failed.
- If `rocqchk` fails to return a result (*i.e.*, exhibits a divergent behavior), evaluators shall report it in the Evaluation Technical Report.

3.3 Features Restrictions

Rocq architecture is highly modular and extensible, and as a consequence provide several interfaces to experts in order to increase its capabilities. Although these interfaces are not supposed to call

Rocq kernel trustworthiness into question, Rocq definitely is a highly complex system, and the impact of changes is often difficult to measure for non-experts.

Typing Rules Rocq's formal system is based on a sound higher-order logic, meaning it is not possible to create a proof of `False` without contradictory hypotheses. However, Rocq also provides several arguments and options which may endanger this key property.

Since at least Rocq 8.0, the `Set` sort is predicative by default. Rocq provides an option to change this behavior (`-impredicative-set`), making `Set` impredicative again. This option notably introduces inconsistencies with respect to certain axioms provided by the standard library.

Developers Rule 3.4. Developers shall not use the `-impredicative-set` command-line argument.

`-type-in-type` is another command-line argument known to make Rocq logic inconsistent. Its purpose is to collapse the universe hierarchy, meaning `Type` becomes impredicative.

Developers Rule 3.5. Developers shall not use the `-type-in-type` command-line argument.

Recent Rocq versions (starting from 8.11) provide vernacular commands to locally disable guard checking, positivity checking and universes checking (by unsetting related flags). Constants defined when these flags are unset are advertised as axioms by Rocq's kernel, but are evaluated nonetheless.

Developers Rule 3.6. Developers shall not unset the type-checking flags `Guard Checking`, `Positivity Checking`, and `Universe Checking`.

Plugins. Rocq extensibility notably relies on a plugin framework to dynamically load external OCaml code. By means of plugins, experts can provide advanced features without the need to patch and build Rocq from source. Many features of Rocq are implemented as external plugins (*e.g.*, extraction, tactic language, decision procedures, etc.). Implementing and reviewing a Rocq plugin quickly requires an important knowledge of the Rocq codebase.

Developers Rule 3.7. Developers can use plugins which are part of the Rocq standard library without any restrictions.

Many well-established third-party libraries implement their core features by means of plugins. In particular, the Rocq installer distributed for Windows (version 8.8.2 and later) contains several well established, actively maintained packages.

Evaluators Rule 3.4. OCaml code provided by third-party libraries shall be considered as actively used by a formal development as soon as they are loaded by Rocq. As a consequence, evaluators shall review them.

Developers Rule 3.8. If `rocqchk` exhibits a divergent behavior, then developers shall not use plugins which are not part of the standard library.

Goal Admissions. The tactic `admit` and the vernacular command `Admitted` allow discarding goals automatically, without the need to actually provide a proof.

Developers Rule 3.9. Developers shall not use the tactic `admit`, and the vernacular command `Admitted` to conclude their proofs.

Feature Conservatism. Not all Rocq features have the same degree of maturity. We favor stable, battle-tested features over experimental and deprecated ones.

Developers Rule 3.10. Developers shall not use features explicitly flagged as experimental in the Rocq user manual.

Developers Rule 3.11. Developers shall avoid using features explicitly flagged as deprecated in the Rocq user manual.

Chapter 4

Development Recommendations

In this chapter, we give a set of rules to write Rocq specifications and proof scripts. Their purpose is to make a Rocq formal development less subject to inadvertent inconsistencies, and easier to evaluate.

4.1 Axioms and Hypotheses

Rocq allows for introducing opaque, well-typed constants which lack a body, that is a term of the expected type. One notable way to achieve this is the `Parameter` vernacular command (or its synonyms, *e.g.*, `Variable`, `Hypothesis`, etc.).

Evaluators Rule 4.1. Starting from version 8.5, the `Print Assumptions` command shall be used in order to determine which constants lacking a definition body have been used to prove key theorems.

We distinguish two use cases for this feature: introducing logical statements which cannot be proven using Rocq's logic, and introducing statements which could be proven or defined using Rocq's logic, but are identified as out of the scope of the model (*e.g.*, environment assumptions, axiomatized operations, etc.). In the rest of this document, logical statements which cannot be proven using Rocq's logic are referred to as *axioms*, and statements identified as out of the scope of the model as *hypotheses*. Vernacular commands like `Parameter` and its synonyms do not differentiate axioms and hypotheses, but they are treated differently nonetheless in the context of Common Criteria evaluations.

Axioms. The introduction of an axiom is delicate, as it may result in making the Rocq logic inconsistent. Besides, certain "standard axioms" are known to be inconsistent with Rocq logic (*e.g.*, the axiom of description together with the excluded middle). However, the standard library of Rocq provides several axioms that can be safely added to Rocq, according to the Rocq FAQ.

That being said, the standard library provides the means to avoid using most of these axioms. For instance, *e.g.*, generalized rewriting can be leveraged in place of relying on the proof irrelevance and functional extensionality axioms.

Developers Rule 4.1. Developers are only allowed to use the logical axioms defined in the `Stdlib.Logic.*` modules of the Rocq standard library. Besides, they shall list the logical axioms they use in their formal development.

Hypotheses. Hypotheses are a necessary concept to model the environment of the Target of Evaluation. Because they are assumed (rather than defined or proven), hypotheses can pose a threat to the consistency of a model. We say hypotheses are contradictory when they can be used to create a proof of `False`, which then can be used to prove anything. Furthermore, the automation features of Rocq can make the use of contradictory hypotheses more difficult to uncover.

One approach to prevent the introduction of inconsistent hypotheses is to prove that there exists at least one model (potentially trivial) which satisfies them. The purpose of this model is only to demonstrate the hypotheses are not contradictory. From this perspective, the model does not need to be similar to the product implementation.

Developers Rule 4.2. Developers shall provide a model —potentially trivial— which satisfies the hypotheses they introduce. They can use a different formal method to that end (*e.g.*, a fully automated one), but in this case the consistency between the statements in Rocq logic and their model shall be justified.

We consider a capability system, where capability objects of type `Cap` grant authorization to perform certain actions of type `Act`. The capability system provides two operations:

- `merge : Cap → Cap → Cap`
- `grant : Cap → Act → Prop`

The capability system is assumed to provide the following guarantees:

- `merge` is symmetric, and associative
- If one of the operands of `merge` authorizes an action, then the result of `merge` authorizes this action
- If none of the operands of `merge` authorizes an action, then the result of `merge` does not authorize this action

The actual implementation of the capability system is likely to be complex, but we can assume there exists one by introducing a set of hypotheses. One possible approach to group a set of hypotheses together is to define them within a dedicated typeclass.

```
Class AccessControl (Cap Act : Type) :=
  {
    merge : Cap → Cap → Cap;
    grant : Cap → Act → Prop;
    merge_sym : forall (c1 c2 : Cap),
                merge c1 c2 = merge c2 c1;
```

```
  merge_assoc : forall (c1 c2 c3 : Cap),
                merge c1 (merge c2 c3)
                = merge (merge c1 c2) c3;
  merge_rule_1 : forall (c1 c2 : Cap)
                   (act : Act),
                 (grant c1 act ∨ grant c2 act)
                 → grant (merge c1 c2) act;
  merge_rule_2 : forall (c1 c2 : Cap)
                   (act : Act),
                 (∼ grant c1 act ∧ ∼grant c2 act)
                 → ∼grant (merge c1 c2) act
}.
```

To show the hypotheses we have introduced are not contradictory with the following model, we provide a minimal model. For instance,

- `Cap` := `nat` and `Act` := `nat`
- `merge` := `max` and `grant` := `(>=)`
- We prove `max` is symmetric and associative
- We prove `(c1 >= act \/ c2 >= act) -> max c1 c2 >= act`
- We prove `(c1 < act /\ c2 < act) -> max c1 c2 < act`

Rocq provides several approaches to introduce and manipulate hypotheses in a clean way, namely by:

- Using the `Class` vernacular command to aggregate them inside a dedicated typeclass.
- Using the `Hypothesis` vernacular command (or its plural form `Hypotheses`) within a `Section` environment.
- Using the `Parameter` vernacular command (or its plural form `Parameters`) within a `Module Sig` environment to be used with functor modules.

Developers can also use the `Parameter` vernacular command (and its synonyms) outside of a module signature in order to introduce their hypotheses. We warn against this practice, since there is no way for Rocq to assert the compatibility between the hypotheses and a model which satisfies these hypotheses in this case. On the contrary, Rocq provides the means to assert such compatibility for the three approaches previously listed.

4.2 Empty Types

In Rocq, nothing prevents developers from defining types which, contrary to their intent, are empty. Consider the following definition of binary trees:

```
Inductive tree (a : Type) : Type :=
| node : a → tree a → tree a → tree a.
```

This definition of `tree` lacks a base case constructor. It is therefore not possible to construct a valid term of type `tree a`. The following type happens to be empty as well, but for a different reason:

```
Definition max_nat := { n : nat | n = S n }.
```

The property `n = S n` is contradictory: there is no such Peano number. Less obvious uninhabited types can remain undetected if they are only manipulated through the `forall` quantifier. In practice, empty types pose a threat similar to contradictory hypotheses.

Developers Rule 4.3. For user-defined types, developers shall prove the existence of at least one inhabitant, or shall justify in depth their reasons not to provide such a proof. We recommend introducing an `Inhabited` typeclass in the first case.

The `Inhabited` typeclass can be defined and used as follows.

```
Class Inhabited (A: Type): Type := inhabit { inhabitant: A }.
Inductive Foo: Type := Bar: nat → Foo.
Instance Foo_inhabited: Inhabited Foo := inhabit _ (Bar 0).
```

Evaluators Rule 4.2. Evaluators shall ensure main theorems do not use logical quantifiers with user-defined types which lack a proof of the existence of at least one inhabitant.

4.3 Coding Style

There is no well-established coding style for Rocq development. The present document does not attempt to fill this gap, but rather to identify and discourage known bad practices.

Standard Library Precedence The standard library is a collection of Rocq library which is distributed with the formal system. By default, the `Init` libraries are loaded automatically by Rocq. The use of the standard library is ubiquitous in the Rocq community.

Developers Rule 4.4. Developers shall take care not to reuse terminology already used by the standard library, to avoid any confusion.

Redefining a custom data-type named `list` is likely to confuse evaluators.

```
Inductive list (a : Type) :=
| Push (l : list a) (x : a)
| Empty.
```

Command	Recommended usages
`Lemma`	Proven formal statements
`Hypothesis`	Logical statements which **can be proven** using Rocq logic, but are hypotheses of the model

Table 4.1: Example of restricted use of vernacular commands

To avoid confusion, use a different name.

```
Inductive sequence (a : Type) :=
| Push (l : sequence a) (x : a)
| Empty.
```

Vernacular Commands Rocq provides many redundant keywords and commands, *e.g.*, `Remark`, `Fact`, `Lemma`, `Theorem`, `Corollary`, `Definition` can be used to prove statements.

Developers Rule 4.5. Developers shall take extra care to use as few vernacular commands as possible. Their use shall remain consistent throughout their formal developments, in order to facilitate understanding of their development (see Table 4.1 for an example).

Sections. Rocq provides a convenient factoring mechanism to avoid code duplication and boilerplate. Within a `Section`, developers can declare local constants (using various keywords, such as `Hypotheses` or `Variable`). If a definition introduced inside the section uses such a constant, the latter becomes a parameter of the former once the Section has been closed. We can illustrate this mechanism with the `lmap` function, as defined in the following snippet:

```
Section lmap.
  Variables (a b : Type) (f : a → b).

  Fixpoint lmap (l : list a) : list b :=
    match l with
    | x::rst ⇒ cons (f x) (lmap rst)
    | nil ⇒ nil
    end.
End lmap.
```

The output of the `Print` command within the section is as follows:

```
lmap =
fix lmap (l : list a) : list b :=
  match l with
  | nil => nil
  | (x :: rst)%list => (f x :: lmap rst)%list
```

```
  end
     : list a -> list b
```

Both the type and the body of `lmap` are identical to our definition. However, using the same command outside of the section gives a slightly different result. Because `lmap` makes use of the variable `f`, `lmap` now requires additional arguments.

```
lmap =
fun (a b : Type) (f : a -> b) =>
fix lmap (l : list a) : list b :=
  match l with
  | nil => nil
  | (x :: rst)%list => (f x :: lmap rst)%list
  end
     : forall a b : Type, (a -> b) -> list a -> list b
```

Sections allow for organizing Rocq developments similarly to pencil proofs. By default, long `Section`s can make predicting the exact type of a definition harder, because the type written by the developer is extended when the `Section` is closed. However, there exists a variant of the `Proof` vernacular command which can be used to list `Section`'s variables used to prove a given statement. For instance, given following snippet:

```
Section test.
  Variable exfalso : False.

  Lemma use_exfalso : ~True.

  Proof using.
    destruct exfalso.
  Qed.
End test.
```

Rocq will output the following error message when processing the `Qed` command.

```
Error:
The following section variable is used but not declared:
exfalso.

You can either update your proof not to depend on exfalso, or you can
update your Proof line from
Proof using
to
Proof using exfalso
```

Developers Rule 4.6. When using the section mechanism to formulate the statement of their main theorems, developers shall make the list of hypotheses used in the proof explicit with the

`Proof using` feature.

Rocq Syntax Extensibility. Rocq provides several features which aim to make writing code easier. The most famous feature is probably the `Notation` vernacular command. `Notation` allows for extending the Rocq parser on-the-fly. Correctly leveraged, this feature can help make formal developments more concise and readable. However, overused, it can have the opposite effect, and it can make a Rocq codebase harder to read or modify. Evaluators of formal developments may want to edit the codebase, *e.g.*, in order to find hypotheses inconsistencies.

Developers Rule 4.7. Notations shall favor regular ASCII characters over Unicode characters, so that they do not require a particular IDE setup to use them.

The `Notation` command allows for fine-tuning in terms of associativity and precedence. In theory, this allows improving the readability of a development, by removing dispensable parentheses. In practice, leaning on notation's precedence and associativity can quickly introduce major ambiguity for the evaluators.

Developers Rule 4.8. Developers shall use parentheses to explicit notations precedence in the following cases:

- The statement of the main theorems, especially if they mix several logical operators (*e.g.*, universal quantifiers, implications, and equivalences).
- Any notations introduced by developers.

Do not assume evaluators know by heart the precedence and associativity level of each notation.

```
Lemma p_and_q_or_q_and_p : P ∧ Q ∨ Q ∧ P.
```

Instead, make them explicit using parentheses.

```
Lemma p_and_q_or_q_and_p : (P ∧ Q) ∨ (Q ∧ P).
```

There is another convenient feature that Rocq provides for developers to reduce the verbosity of their development: implicit coercions. A coercion is a function of type `A → B` that Rocq will be able to insert automatically *and* implicitly in Gallina terms where a term of type `A` is used in place of an expected term of type `B`.

We consider the function `of_nat`, which maps `0` to `false` and any other natural number to `true`.

```
Definition of_nat : nat → bool :=
  fun x ⇒ negb (Nat.eqb x 0).
```

We can tell Rocq to use this function as a coercion from `nat` to `bool`, using the `Coercion` command.

```
Coercion of_nat : nat >-> bool.
```

Afterwards, we can use `nat` terms wherever we would have to use `bool`.

```
Definition nat_neg : nat → bool := negb.
```

Coercions are implicit by default, meaning that we will not be able to see them when we inspect the model (*e.g.*, using the `Print` command):

```
Print nat_neg.
```

```
nat_neg = fun b : nat => negb b
     : nat -> bool
```

Fortunately, it is possible to tell Rocq we want coercions to be *explicit*:

```
Set Printing Coercions.
```

Printing `nat_neg` this time gives the following output:

```
nat_neg = fun b : nat => negb (of_nat b)
     : nat -> bool
```

Developers Rule 4.9. Developers shall justify their uses of implicit coercions, in particular with respect to their impact on the readability of statements of the main theorems.

Evaluators Rule 4.3. Evaluators shall use the `Set Printing Coercions` vernacular command when they review the model.

Proof Handling A key concept of Rocq is that proofs are regular terms that happen to live in a particular sort (`Prop`). To ease the definition of such terms, Rocq provides an interactive environment along with a dedicated language called Ltac. Ltac's so-called proof scripts are translated into Gallina terms and then type checked. For instance, the `induction` tactic is translated into the application of the proofs of the subgoals generated by the tactic to the induction principle of the term passed as an argument. Similarly, the `destruct` tactic is typically translated into a `match` statement. The `Print` command can be leveraged in order to illustrate that mechanism.

```
Inductive even : nat → Prop :=
| even_0 : even 0
| even_S_S_n (n : nat) (even : even n) : even (S (S n)).

Lemma even_n_or_even_S_n (n : nat)
  : even n ∨ even (S n).
Proof.
  induction n as [ | n IHn ].
  + left; constructor.
  + destruct IHn as [IHn|IHn].
    * right.
```

```
    now constructor.
  * now left.
Qed.

Print even_n_or_even_S_n.
```

will output:

```
even_n_or_even_S_n =
fun n : nat =>
nat_ind (fun n0 : nat => even n0 \/ even (S n0)) (or_introl even_O)
  (fun (n0 : nat) (IHn : even n0 \/ even (S n0)) =>
   match IHn with
   | or_introl IHn0 => or_intror (even_S_S_n n0 IHn0)
   | or_intror IHn0 => or_introl IHn0
   end) n
     : forall n : nat, even n \/ even (S n)
```

It is possible to write the definition of `even_n_or_even_S_n` directly in Gallina, but in practice proofs terms tend to quickly become of large size. Also, the resulting Ltac proof script is much closer to what a pencil proof of the same statement would look like, compared to the generated term.

Developers Rule 4.10. Developers shall take care to carefully organize their proof scripts to make them readable and reviewable. In particular, they shall use goal markers (*e.g.*, `+`, `-`, or `*`).

A "flat" proof script is hard to read.

```
induction n as [ | n IHn ].
left;constructor.
destruct IHn as [IHn|IHn].
right.
now constructor.
now left.
```

Using appropriate markers, the structure of the proof becomes clearer and the resulting proof script easier to read.

```
induction n as [ | n IHn ].
+ left;constructor.
+ destruct IHn as [IHn|IHn].
  * right.
    now constructor.
```

```
  * now left.
```

Developers Rule 4.11. Developers shall enclose their proof scripts with appropriate vernacular commands (*e.g.*, `Proof`, `Next Obligation` on the one hand, and `Qed` or `Defined` on the other hand) even when this is not strictly required by Rocq.

Rocq allows for writing a proof script directly after a lemma statement, but this is considered a bad practice.

```
Lemma modus_ponens (P Q : Prop) (p : P) (imp : P → Q) : Q.
  apply imp.
  exact p.
Qed.
```

The vernacular command `Proof` shall be used to explicit the entry point of a proof script in this context.

```
Lemma modus_ponens (P Q : Prop) (p : P) (imp : P → Q) : Q.
Proof.
  apply imp.
  exact p.
Qed.
```

Note that, in addition to these requirements, developers should consider keeping the size of their proof scripts manageable. In practice, a long proof script (*e.g.*, more than 100 lines) can be reduced by isolating intermediary results which can be proven as auxiliary lemmas. Conforming to this approach can greatly improve the readability of a formal development, and therefore ease the work of evaluators.

Hints. Rocq provides several features to enable *proof automation*, such that the theorem prover tries to construct proof terms through "heuristics" (provided by the Rocq distribution or by developers). To add new "heuristics," the `Hint` family of vernacular command is leveraged:

- `Hint Resolve my_lemma` will make certain tactics (*e.g.*, `auto`) try to apply `my_lemma` to progress in the current goal resolution
- `Hint Rewrite my_equation` will make the `auto_rewrite` tactics try to use `my_equation` to progress in the current goal resolution
- `Hint Extern my_ltac` will make certain tactics (*e.g.*, `auto`) try to use the tactics `my_ltac` to solve the current goal

The use of `Hint`, albeit very convenient, can have a significant impact over Rocq's performance, and make the resulting proof's shape hard to predict.

Developers Rule 4.12. Developers shall always assign their hints to a hint database, preferably dedicated to their project although this is not mandatory (*e.g.*, they can be added to the `core` database if needed).

Metaprogramming. Since Rocq 8.5, it is possible to use Ltac within a Gallina term. This enables powerful metaprogramming development. Indeed, the same Ltac script may generate totally different Gallina terms depending on the goal to solve.

Developers Rule 4.13. Developers shall only use `ltac:(...)` with a tactic. The tactic name shall be carefully chosen to be self-explanatory.

```
Definition test_nat : nat :=
  ltac:(match goal with
        | ⊢ nat ⇒ exact 0
        | ⊢ bool ⇒ exact true
        end).
```

```
Ltac choose_default :=
  match goal with
  | ⊢ nat ⇒ exact 0
  | ⊢ bool ⇒ exact true
  end.

Definition test_nat: nat :=
  ltac:(choose_default).
```

Consistent Naming. While variables can be arbitrarily named when introduced, it is sensible to keep a consistent naming scheme in order to help evaluators reviewing the code. Types can also be aliased, care should be taken about which name to use given the context in the code.

Developers Rule 4.14. Developers shall use a consistent naming scheme with regards to typing when introducing variables.

Suppose that a user-defined type `permission` is introduced.

```
Lemma perm_eq_dec: forall (p op: permission),
```

```
  { p = op } + { p <> op }.
```

The variable `op` could be confused with a variable of another type (e.g., `operation`) at first glance.

```
Lemma perm_eq_dec: forall (p1 p2: permission),
  { p1 = p2 } + { p1 <> p2 }.

Lemma perm_eq_dec': forall (p p': permission),
  { p = p' } + { p <> p' }.
```

Developers Rule 4.15. Developers shall take care to use the right name when adding type annotations with regards to aliases.

```
Definition error_raised: Type := bool.

Definition trace_has_error (l: list error_raised): bool :=
  List.fold_left (fun (acc b: bool) ⇒ acc || b) false.
```

```
Definition error_raised: Type := bool.

Definition trace_has_error (l: list error_raised): bool :=
  List.fold_left (fun (acc: bool) (b: error_raised) ⇒ acc || b) false.
```

Type Annotations. While Rocq can infer types, adding type annotations to definitions and theorem statements can significantly help evaluators reviewing the code.

Developers Rule 4.16. Developers shall add type annotations to all definitions, lemma and theorem statements.

```
Definition has_true l :=
  List.fold_left (fun acc b ⇒ acc || b) false.

Lemma has_true_iff:
  forall l,
```

```
    (has_true l = true ↔ exists n, List.nth_error l n = Some true).
```

```
Definition has_true (l: list bool): bool :=
  List.fold_left (fun (acc b: bool) ⇒ acc || b) false.

Lemma has_true_iff:
  forall (l: list bool),
    (has_true l = true ↔ exists (n: nat), List.nth_error l n = Some true).
```

Unused Function Parameters. Unused function parameters can be misleading, and must be avoided as much as possible.

Developers Rule 4.17. Unused parameters in function definitions must be avoided as much as possible, or, otherwise, they must be explained and made explicit by the use of an underscore.

```
Definition incorrect_thunk_one (x: unit) := 1%nat.
```

```
Definition correct_thunk_one (_: unit) := 1%nat.
```

4.4 Code Structure

Dependency graph In order to make evaluation and reviewing of Rocq developments easier, we recommend to provide evaluators with a dependency graph of definitions, lemmas, theorems, etc using for instance `coq-dpdgraph`[1]. This could be done in addition with describing the files to help evaluators navigating them.

[1] https://github.com/rocq-community/coq-dpdgraph