

Notations of the Four Colour Theorem proof

This document is primarily a reading guide for the Coq proof scripts of the Four Colour Theorem proof. It is divided in two parts:

- The first part should be useful for reading the coqdoc outline of the proof, that is, the mathematical definitions, the programs, and the statements of lemmas and theorems, but not their proofs (i.e., not the tactic scripts). We describe the basic definitions that are used throughout the development, the naming convention of lemmas, and the notations (syntax extensions) that we used.
- The second part describes the tactic shell used for writing the proof scripts, and is therefore intended for those interested in *how* the individual lemmas are proved. Apart from this immediate use, the tactic shell description should also be interesting to those wishing to port it to other proof systems in order to use the style of proof developed for the Four Colour theorem on other projects.

Definitions and lemma statements may be browsed passively, but the tactic scripts will be best viewed in an interactive proof environment such as ProofGeneral. This will allow to

- track and display the cumulative effect of bookkeeping, rewriting, and backward chaining commands on the proof state
- to break long command lines (e.g., chain of rewrite steps) to display intermediate states of the proof
- insert new commands to further examine the proof state, e.g., pretty-printing a value

The grammar and printing rules should on the other hand be skipped entirely, as they are entirely subsumed by the present document, in addition to being highly technical and rather obscure. In particular, the file [tacticext.v](#), which consists almost exclusively of such rules, can safely be skipped.

1 Formalism, conventions and notations

The main obstacle to overcome in the formalization of a fairly large proof such as that of the Four Colour theorem is complexity. Simply following the common wisdom of breaking up large proofs into small lemmas would have resulted in a large, haphazard, and unmanageable collection of technical lemmas. Much of our efforts were therefore geared towards limiting the size of this collection. We accomplished this partly by using forward reasoning to embed many purely technical lemmas inside the proof of more meaningful results, and we developed the tactic shell described below to deal more effectively with the larger scripts that resulted.

Still, technical lemmas are a necessary evil: three out of four of the 2500 lemmas in the proof are small technical results whose proof is under five lines. This is a sizeable number, so a significant part of the proof effort went into designing and redesigning the basic formalism so that the collection of technical lemmas remained manageable. For instance, we went through five different definitions of the basic notion of *paths* before settling with the current design, even though we only moved to a new definition when it became obvious that the current one was stalling us.

As is usual in software development, the key to keeping the proliferation of lemmas under control was orthogonality. We found that we could cut down dramatically the number of lemmas by using of definitions and equations to relate the different notions involved in the proof; this allows us to transfer properties from one notion to the other, so that we only need to state the property once. For example, we define paths in such a way that the statement “a sequence p is a path starting at x ” is *identical* to “there is an edge from x to y , and the rest of p is a path from y ”, and we also show that the the statement is *equal* to “ q is a path starting at x , and r is a path starting at the last item in q ” when p is the concatenation of q and r . Thus, simply by expanding definitions and rewriting, we can reduce any form of path associativity of the associativity of the “and” connective: we don’t need any path associativity lemma.

The reduction in the number of lemmas obtained by omitting definitional and equational variants of basic lemmas is significant, and grows non-linearly with the size of the formalization because many of these lemmas are themselves equations. The downside is that purely syntactic lemma application usually fails, rendering the `Auto` tactic rather ineffective. Many of the features of the tactic shell are designed to counter this, by making manual application more efficient.

1.1 Basic formalism

1.1.1 Boolean propositions

We use Coq’s *coercion* feature to inject the two-element concrete datatype `bool` into the sort `Prop` of logical propositions. For example, the goal

```
(andb (leq n (5)) (negb (odd n)))
```

is really notation for the equation

```
(andb (leq n (5)) (negb (odd n))) = true
```

Using booleans as propositions provides a very effective way of performing propositional reasoning, by evaluating and/or rewriting boolean predicates and connectives. We use this technique extensively: most of the predicates we define on datatypes return a `bool` rather than a `Prop`. For example we use the boolean predicate `leq` to compare integers, rather than the inductively defined dependent logical predicate `le` from the Coq standard library. This has many benefits:

- All definite comparisons are decided by computation (e.g., `(leq (3) (5))` or `(leq (4) (2))`).
- Base cases `(leq (0) n)` and `(leq (S n) (0))` are resolved by evaluation
- The most common inductive case `(leq (S m) (S n)) = (leq m n)` is resolved by evaluation
- There is no need to perform inversion on the predicate when we have to do an induction on some other quantity than the right-hand side of the comparison – that is, most of the time.
- If needed, the disjunction hard-coded in the `le` inductive predicate can be exposed by rewriting with the tactic `Rewrite leq_eqVlt`.

These benefit come in part from the switch from an inductive to a recursive definition; however, they also come from exploiting two key properties of boolean predicates:

- Boolean predicates naturally allow classical reasoning, using the `Discriminate` tactic to generate a contradiction from the equality `false = true`. This allows us classical reasoning throughout, even though Coq uses

intuitionistic logic (we only need depend on an excluded middle assumption in the small portion of the proof that involves real numbers).

- The logical equivalence of boolean predicates translates into the (intensional) equality of their value, and such equalities can be used for rewriting any part of an assumption or goal, e.g., using

```
Lemma leq_eqVlt : (m, n : nat)
  (leq m n) = (orb m =d n (ltm m n)).
```

Nevertheless, it is also necessary to be able to revert to a logical proposition (in `Prop`) in many cases: to interface with the Coq standard library, to apply primitive tactics such as decomposition (intro patterns in particular), introduction (for universally quantified proposition), or rewriting (for intensional equalities). We use the predicate `reflect` to declare the connections between booleans and propositions:

```
(reflect P b)
```

means that `P` and `b = true` are equivalent. For instance we prove

```
Lemma andPx : (b1, b2 : bool) (reflect b1 /\ b2 (andb b1 b2)).
Lemma lePx : (m, n : nat) (reflect (le m n) (leq m n)).
```

along with notations such as `Syntactic Definition andP := (andPx ? ?)`.

To facilitate the use of such reflection properties we prove a set of introduction and elimination lemmas such as

```
Lemma elimT : (P : Prop; b : bool) (reflect P b) -> b -> P.
Lemma introN : (P : Prop; b : bool)
  (reflect P b) -> ~P -> (negb b).
```

We can use these lemmas directly, for example starting a proof of `(negb (andb b1 b2))` with

```
Apply (introN (andPx b1 b2)); Intros [Hb1 Hb2].
```

and then deriving a contradiction from the assumptions `Hb1 : b1` and `Hb2 : b2`, or converting a “boolean” assumption `Hmn : (leq m n)` to an equivalent inductive form `Hmn : (le m n)` with

```
Assert Hmn' := (elimT leP Hmn); Clear Hmn; Rename Hmn' into Hmn.
```

However, most of the time we use the [view feature](#) of our tactic shell to perform such operations, because it provides a much more compact, readable, and modifiable notation. The two examples above can be written

```
Apply/andP=> [[Hb1 Hb2]].
Move/leP: Hmn => Hmn.
```

Also, the `elimT` lemma is declared as coercion, so one can write `(leP Hmn)` instead of `(elimT leP Hmn)`.

Because `reflect` is defined as an inductive dependent predicate, we can also use direct case analysis to mimic classical reasoning and split a proof in two according to a predicate. For example,

```
Case (lePx m n); Intros Hmn.
```

generates two subgoals, in which the boolean expression `(leq m n)` has been replaced by `true` (resp. `false`), and which have a new assumption `Hmn : (le m n)` (resp. `~(le m n)`). However more often we find it more convenient to analyze then boolean value, using the [equality generation](#) feature of our tactic shell to create boolean assumptions, e.g.,

```
Case Hmn: (leq m n).
```

creates two branches, instantiates `(leq m n)` in the goal, and adds a new assumption `Hmn : (leq m n) = true` (resp. `false`); recall that the former is convertible to `Hmn : (leq m n)`.

We should stress that it is often the case that the same boolean expression has *several* different propositional interpretations. A trivial example of this is n -ary boolean connectives. We define inductive 3, 4, and 5-way

conjunctions in `Prop`, and prove reflection lemmas such as

```
Lemma and3Px : (reflect (and3 b1 b2 b3) (and3b b1 b2 b3)).
```

where `(and3b b1 b2 b3)` is notation for the tree of “and”s (`andb b1 (andb b2 b3)`). This allows us to decompose a 3-way boolean conjunction with a single application of reflection. But we can also use `andP` on the same expression (e.g., if we only need the first conjunct). Conversely, the expression doesn’t have to be entered as a 3-way “and”; the tree form can (and often does) arise from partially evaluating and/or rewriting the expression.

For a more elaborate example, in our library of operations on [sequences](#), we define a boolean predicate `has` that, given a boolean predicate `p` and a sequence `s` checks whether `s` contains some `x` such that `(p x)` holds. We provide two reflection lemmas for `has`:

```
Lemma hasPx : (reflect (EX x | (s x) & (p x)) (has p s)).
```

```
Lemma has_subPx :
```

```
(reflect (EX i | (lt i (size s)) & (p (sub s i))) (has p s)).
```

The first lemma is used to get the element `x`, the second one to get the index at which `x` occurs in `s`. We also prove a reflection lemma for the negation of `has`:

```
Lemma hasPnx :
```

```
(reflect (x : ?)(s x) -> (negb (p x)) (negb (has p s))).
```

The `view` feature allows us to use any of these lemmas to an assumption or goal involving `has`; e.g.,

```
Apply/hasPn=> [Hs].
```

will start a proof by contradiction of `(has p s)`, introducing the assumption that the negation of `p` holds everywhere in `s`. There is more however, as the definition of `has` is computational: if `s` is defined as the sequence `(Seq x y & s')` that starts with `x` and `y`, then `(has p s)` evaluates to

```
(orb (p x1) (orb (p x2) (has p s')))
```

and therefore we can also use the reflection lemmas `orP` and `or3P` to decompose `(has p s)`. Furthermore, we can also use all these lemmas for more specialized predicates that are defined in terms of `has`, such `has fband`, which computes the closure of a ring of darts in a hypermap, under the “face” permutation. Even without adding any new lemmas to the development, we have many ways of establishing and using statements involving `fband`. This orthogonality is the key to keeping down both the number of lemmas and the total size of the proof.

1.1.2 Equality datatypes

Next to the logical connectives, the logical predicate for which we most often need boolean reflection is the inextensional (Leibniz) equality used by Coq’s rewriting tactics. Since equality is not always decidable in Coq’s intuitionistic setting, equality reflection is not uniformly available. We therefore use Coq’s record and coercion features to define a `dataSet` as a `Set` with a boolean predicate that reflects equality, denoted ‘`eqd`’ (or ‘`=d`’ when used infix, as in the `leq_eqvlt` lemma above). We define canonical `dataSets` for all the basic datatypes involved in the proof – booleans, signed and unsigned integers, colours, chromograms, parts, etc., as well as for all the basic datatype constructions: sequences, sums, products, comprehensions, etc. All the hypermap theory that we develop assumes that the underlying set of darts has a `dataSet` structure.

A default `dataSet` structure can be defined for `sets` for which equality is decidable, and the latter can be established using Coq's `Decide Equality` tactic. However we generally find it preferable to supply an explicit version of the equality predicate `eqd`, as this allows us to control its computational behavior.

Given a `dataSet` `d`, it makes sense to identify (enumerable) subsets of `d` with boolean predicates over `d`, as we can use the `eqd` predicate to define finite enumerations (singletons, pairs, etc) over `d`. We therefore define

```
Definition set [d : dataSet] : Set := d -> bool.
```

along with the usual array of set operators (e.g., `setU`, `setI`, `setD`, respectively set union, intersection and difference). We do not, however, prove de Morgan laws for such sets, as they would be redundant with the corresponding boolean laws.

1.1.3 Sequences

As can be expected sequences are ubiquitous in the combinatorial part of the proof, both in the computational part, where we use everything from bit strings to two-dimensional arrays of decision diagrams, and in the graph theory part where sequences are used as a uniform representations for [paths](#). We define the type `(seq d)` of sequences over a `dataSet` `d`, since many sequence operations require the equality predicate; in particular we declare the membership predicate as a coercion `mem : seq ->> set`; this allows us to use sequences as finite sets throughout the proof.

We define a comprehensive library of operations on sequences, which covers most of the operations found in a good Lisp library, and we provide lemmas to handle the interaction between most pairs of operations: e.g.,

```
Lemma size_cat : (s1, s2 : (seq d))
  (size (cat s1 s2)) = (addn (size s1) (size s2)).
```

gives the rule for computing the length of the concatenation of two sequences. Like `size_cat`, most such lemmas are unconditional rewrite rules.

We model partial operations such as indexing (`sub`) by supplying a default value as an additional argument, always passed first. The `(last x s)` operation, which returns the last element of `s`, defaulting to `x` if `s` is empty, can thus equivalently be seen as returning the last element of the non-empty sequence `(Adds x s)`.

1.1.4 Paths

Paths are the central concept of the graph theory part of the proof. After exploring several dead ends involving dependent records, we decided to model paths as non-empty sequences constrained by a boolean predicate. Specifically, an `e`-path, for a binary boolean predicate `e` on a `dataSet` `d`, consists in a starting point `x`, a (possibly empty) continuation sequence `p`, and a proof `Hp` that every successive pair `y,z` of elements in the sequence `(Adds x p)` verifies `(e y z)`. The latter condition is described by a boolean predicate `path` defined by structural recursion on `p`. We always pass the three components of a path (`x`, `p`, and `Hp`) separately, rather than packaging them in a dependent record: most of the time `x` and `p` are implicit

parameters inferred from the type of H_P , so for most practical purposes H_P can be used as if it were a record.

Because paths are explicitly defined in terms of sequences, we can use all the operations of the `seq` library to cut and splice paths, using the following lemma to rewrite the `path` assumption H_P .

```
Lemma path_cat : (d: dataSet; e: (rel d); x: d; p1, p2: (seq d))
  (path x (cat p1 p2)) = (andb (path x p1) (path (last x p1) p2)).
```

Because we quite frequently need to split a path at one of its points in our proofs, we prove a sequence-splitting lemma

```
Lemma splitPr : (d: dataSet; x: d; p: (seq d))
  (p x) -> (splitr x p).
```

where `(splitr x p)` is a special-purpose dependent inductive that asserts that `p` is of the form `(cat p1 (Adds x p2))`. Thus the command

```
Case/splitPr: {p}Hx => [p1 p2]
```

will use the assumption $(Hx : (p x))$ to introduce new variables `p1` and `p2`, and replace `p` with `(cat p1 (Adds x p2))` in the goal, erasing both Hx and `p`; there's a variant that replaces only selected occurrences of `p`, and saves the equation `p = (cat p1 (Adds x p2))`. Note that this facility is orthogonal to paths: after splitting the sequence, it is usually necessary to use lemma `path_cat` and the reflection lemma for `andb` to split the `path` assumption.

Another common operation is to use the `maps` functional on sequences to lift a (partial) graph morphism to a path morphism. We provide a lemma to rewrite the `(path .. (maps ..))` combination generated by this operation. The morphism can be the identity function: in that case we are using the *same* sequence to represent paths for two different relations.

1.1.5 Finite sets

Most of the intermediate results in the proof of the Four Colour Theorem are combinatorial lemmas on finite datatypes. Such datatypes enjoy many properties that simplify reasoning considerably; in particular, most set-theoretic properties such as inclusion, cardinality, and reachability are computable, i.e., there are reflection lemmas for these properties. We simply define finite datatypes (`finSet`) as equality datatypes (`dataSet`) with an explicit list of all their values. However, we almost never refer to this list directly in proofs, except for basic lemmas on inclusion and cardinality, for defining finite products, sums and comprehensions, and a choice function.

We use the latter to enumerate finite quotients, e.g., to count nodes, edges and faces in a hypermap. We use adjunctions to generate relations between the enumerations of two such quotients; the proof proceeds by lifting an adjunction between two equivalence relations to bijection between the corresponding quotients.

1.1.6 Geometry

The combinatorial object we use to represent maps is called a *hypermap*. A hypermap is basically a pair of arbitrary permutations on a finite datatype, but we prefer to use a slightly more explicit representation: a triple of functions, called *edge*, *node*, and *face*, whose composition is the identity. This makes the geometric interpretation of the

combinatorial structure more apparent and provides a natural six-fold symmetry that we use to cut down on the number of definitions and lemmas. We define about a dozen basic geometric properties on hypermaps: e.g., a *plain* hypermap has only binary edges, a *cubic* hypermap has ternary nodes, the faces of a *pentagonal* hypermap have at least five sides, a *planar* hypermap satisfies the (generalized) Euler formula, etc. We use the record and coercion features of the Coq vernacular to define the different combinations of properties that are used in various parts of the proof. e.g., the configuration occurrence test is valid for a `plain_cubic_pentagonal` map.

1.1.7 Numbers

Since the full statement of the Four Colour Theorem uses the real plane topology, its proof requires a full number hierarchy, from the natural numbers to the reals. The natural numbers are the most heavily used in the proof. We use the standard type `nat`, but redefine almost all the arithmetic operations and predicates in order to make better use of computational reflection: we use `addn` rather than `plus`, `subn` rather than `minus`, `leq` and `ltn` instead of `le` and `lt`, and define canonical `dataSet` structure on `nat`; we also define the natural coercion from `bool` to `nat`.

We use our own datatype `znat` for signed integers; it is mostly used for the branch-and-bound computation in the unavoidability part of the proof. We define a type `frac` for rational numbers but only use it as a stepping stone for constructing a model of the real number structure.

The latter is needed because we use C-Corn's axiomatic-constructive approach, rather than the purely axiomatic approach used in the standard Coq v7 library: we define a `real_structure` as a carrier type with order (`leqr`, `supr`) and arithmetic (`addr`, `oppr`, `mulr`, `invr`) operations,

and a `real_model` as a `real_structure` that satisfies the axioms of a complete ordered field (we also provide the usual infix arithmetic syntax). We then prove the Four Colour Theorem for an arbitrary `real_model` and show separately that such a model can be constructed (assuming the excluded middle axiom) and is unique up to isomorphism.

The construction is the usual Dedekind cut construction; this forces the carrier to be in `Type` rather than `Set`, and more importantly prevents us from using intensional equality in the real axioms. This means we have to use the setoid tactics to rewrite real arithmetic identities; unfortunately these tactics do not support parametrized setoids in Coq v7. This means that in the two files where we use setoid rewriting, we need to locally construct a specific setoid and use explicit rewrite rules to put real arithmetic expressions in a form that exposes the setoid; thus we have about 80 lines of boilerplate in each file defining operations `leqR`, `addR`, etc, and rewrite rules `leqRI`, `addRI`, etc to introduce them, as well as a generic wrapper lemma `rwR` to turn real arithmetic identity into setoid equalities.

1.2 Naming conventions

To keep on top of the 2500 lemmas in the proof we followed simple and systematic naming convention. Most of the time the name of a lemma can be read off its statement: a lemma named `fee_fie_foe` will say something about `(fee .. (fie .. (foe ..) ..) ..)`, e.g. lemma `size_cat` above. We often use a one-letter suffix to

resolve overloaded notation, e.g., `addn`, `addb`, `addc`, `addr` denote `nat`, `boolean`, `color`, and `real` addition, respectively. If a lemma involves several such operations, we only indicate the suffix for the first one, e.g., `Lemma subn_sub`, but we double the suffix to indicate an idempotence or reflexivity law, e.g., `orbb` or `ltnn`. We use `l/r` suffixes to distinguish left and right-handed versions, e.g., `subn_addl`, and `2` to indicate double occurrences, e.g., `leq_add2`.

General correctness lemma whose statement is more complex are simply named after the function analysed, e.g., `cfctr_correct`. Reflection lemmas and rewrite laws are heavily used so they use a specific set of suffix conventions described below. Finally, a handful of theorems have a historical name, e.g., `birkhoff` or `fourcolor`.

Most of our suffixes are single capital letters. We use suffixes for the common operator rewrite laws: commutativity (`C`), associativity (`A`), commutativity under associativity (`CA`), and inverse laws (`_inv`); the *prefix* `S` indicates a symmetry law. The suffix `w` indicates a weakening lemma, usually for a strict inequality, e.g., `ltnw` : $(m, n : \text{nat}) (ltn\ m\ n) \rightarrow (leq\ m\ n)$. The suffixes `I` and `E` indicate an introduction equation or lemma, respectively, while the suffix `_def` indicates a definitional equation or property; the *prefix* `I` indicates an injectivity lemma (e.g., `Iface`).

The suffix `P` indicates a reflection lemma, usually with all dependent parameters implicit; the suffix `Px` is used for the corresponding explicit form (when needed). The suffix `Pn` is used for reflection lemmas of negated predicates, e.g., `hasPn` above.

The suffixes `_sym` and `_trans` indicate symmetry and transitivity lemmas for relations; for reflexivity we usually just repeat the type suffix (e.g., `leqnn` or `eqrr`). Implicit-argument version of the standard lemmas for the Leibniz equality, named `erefl`, `esym`, and `etrans`, are used throughout. Similarly, we use an implicit-argument version `congr` of the standard single-argument function congruence lemma `f_equal`.

The suffixes `T`, `F` and `N` denote `true`, `false`, and `negb` in the reflection introduction and elimination lemmas (e.g., `introN`), and boolean laws (e.g., `orbT`).

Local assumptions and section hypotheses are indicated by the prefix `H` (however we avoid the names `H/H0/H1...` that could interfere with Coq's automatic name generation); we also use `E` for equations and `D` for definitional equations. This is usually just followed by the name of the main variable(s) of the assumption, e.g., `Hx`. Almost all induction hypotheses are called `Hrec`.

1.3 Notations

We use notations sparingly in the Four Colour Theorem proof because although notation can make definitions and statements more readable, they tend to interfere with the proof scripting, and in particular with rewriting and subterm selection. For example, we don't use infix notation for booleans, interger, or color expressions. In many cases (signed integers, sequences, parts) we do provide special notation but by default we always print out the raw form. There is a generic way of forcing pretty-printing in those cases: applying the `pretty` function.

```

Eval Compute in (Znat -3).
= (Zneg (2)) : znat
Eval Compute in (pretty (Znat -3)).
= -3 : (pretty_printed znat)

```

Values of list-like datatypes such as `seq` or `part` are automatically pretty-printed when they have a fixed length.

```

Eval Compute in (Adds true seq0).
= (Seq true) : (seq boolData)

```

We systematically use notation to force the insertion of the implicit argument of unary functions such as `size`, `face`, etc. This allows us to pass these functions to functionals as we would in ML, e.g., `(maps size s)` computes the sequence of sizes of a sequence of sequences. We use the notation `!face` to refer to the fully polymorphic function.

1.3.1 Pattern test

We use a special form for asymmetrical pattern-matching:

```

if term is pattern then match_value else default_value

```

Almost all pattern-matching on sequences use this form; for example, here is the definition of `has`:

```

Variables d : dataSet; a : (set d).
Fixpoint has [s : (seq d)] : bool :=
  if s is (Adds x s') then (orb (a x) (has s')) else false.

```

There is a special case of this form for the choice function on finite datatypes, `pick`: for example we write

```

if pick x in a then (order f x) else (0)

```

instead of

```

if (pick A) is (Some x) then (order f x) else (0)

```

Because of the limitations of the v7 pretty-printing, the `is..in` notation is only pretty-printed correctly if the pattern is of the form `(Some x)`. This is not too much of a problem because Coq will usually not unfold definitions using the `is..in` notation (like `has` above) unless the pattern-matching can be resolved (and then the `is..in` disappears).

1.3.2 Equality

In Coq's intuitionistic formalism one needs to manipulate several notions of equality. Coq v7 only provides notation for intensional (Leibnitz) equality in the `Set` and `Type` universes (`=` and `==`, respectively). We add notation for six other equalities that are used throughout the proof:

- `a =1 b` and `e =2 e'` : extensional equality for unary and binary functions in `Set`, respectively.
- `g =m g'` : extensional equality on hypermaps.
- `x =d y` : boolean equality predicate for equality datatypes. We systematically use this notation for the canonical equality predicate of basic datatypes such as `nat`, `bool`, `color`, etc.
- `x =P y` : reflection for `x =d y`: i.e., `(x =P y) : (reflect x = y x =d y)`.
- ``x = y`` : extensional equality for real numbers; the `Setoid` module is usually required to use this.

1.3.3 Arithmetic

The only new notation for integers is for the “less than” predicate: $(\text{lt}n\ m\ n)$ is notation for $(\text{leq}\ (S\ m)\ n)$. That notation is systematically used for displaying: $(\text{leq}\ (3)\ n)$ will be displayed as $(\text{lt}n\ (2)\ n)$. This convention alone almost halves the number of lemmas in the integer library. One should take this notation into account when specifying occurrences: in the type of $(\text{lt}nS_n\ n) : (\text{lt}n\ n\ (S\ n))$, there are *two* occurrences of $(S\ n)$.

For signed integers we only provide notation for literals: $(\text{Znat}\ m)$ denotes the signed integer m ; as illustrated above, this notation is not used by the default display. There are no specific notations for rational fractions.

We do however provide a complete infix notation for real expressions enclosed in backquotes, e.g., ``x + y - 2/3 * z``. We follow the usual mathematical conventions: binary operators are left-associative, except `/`, which binds tighter than `*`; the sign operator (unary `-`) binds tightest.

``x - y`` is just notation for ``x + -y``; likewise ``x / y`` is notation for ``x * 1/y`` when x is not 1. As for integers, all variants of comparison are just notations for instances of ‘`<=`’, e.g., ``x < y`` is notation for `~`y <= x``: this divides by four the number of arithmetic lemmas in the standard library for reals.

1.3.4 Sequences, configurations, and parts

The notation for the monomorphic empty sequence is `seq0`. The notation $(\text{Seq}\ x_1\ x_2\ x_3\ \&\ s)$ stands for the sequence $(\text{Adds}\ x_1\ (\text{Adds}\ x_2\ (\text{Adds}\ x_3\ s)))$; if s is `seq0` then the final ‘`&s`’ can be omitted. Sequences that end with `seq0`, as well as sequences that start with at least three `Adds`, are displayed with this notation; other sequences get the default display.

Configuration construction programs are sequences of steps: $(\text{Cprog}\ H\ 3\ K\ -1\ Y)$ is notation for $(\text{Seq}\ \text{CpH}\ (\text{CpR}\ (3))\ \text{CpK}\ \text{CpR}'\ \text{CpY})$; this notation is only used for pretty-printing. The notation for configurations is similar: $(\text{Config}^*\ 10\ H\ 6\ H\ 1\ Y\ 5\ H\ 1\ Y)$ stands for

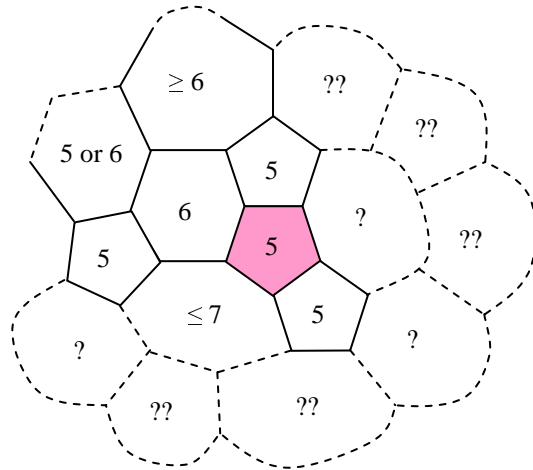
```
(ConfigRecord true (Seq (10)) (Cprog H 6 H 1 Y 5 H 1 Y)).
```

Configurations with a `cfprog` field of the form $(\text{Adds}\ \text{CpH}\ \dots)$ are always displayed with this notation; this covers the 633 explicit configurations used by the proof.

Parts are graph patterns describing the local topography of a pentagonal map; more precisely, a part specifies ranges for the number of sides of each face in the two concentric rings surrounding a given face in the map. The representation of parts is fairly optimised, so we provide a more unified notation that makes both the discharge rule data and the output of the unavoidability scripts much more readable. The term

```
(Part 5 [6+ 5:6] 6 [5] -7 5 *)
```

denotes the following pattern:



where the central face has been coloured. The ? indicates faces whose number of sides is unknown, the ?? faces in the second ring which might not even exist. Note that in the term above, the ranges between square brackets describe segments of the second ring, and are omitted for segments of which nothing is known; in contrast, ranges are given for each of the five neighbours of the central face (the last * indicates that nothing is known about the face adjacent to three pentagons).

1.3.5 Hubcaps and unavailability scripts

A hubcap specifies a series of bounds on the charge that may be exchanged between the central face and its neighbours during the “average number of sides” computation; it is expressed in tenths of sides. The term

(Hubcap T[1;2] <= 0 & T[3;4] <= -6 & T[5] <= -4)

specifies a bound of 0 on the sum of the charges exchanged with the first and second neighbors, of -6 on for the charge exchanged with the third and fourth, and -4 for the charge exchanged with the fifth neighbour.

Hubcaps are explicitly specified in the unavailability scripts; the command

`Hubcap T[1;2] <= 0 & T[3;4] <= -6 & T[5] <= -4.`

proves that a part is successful by checking that it satisfies the constraints above.

There are only four other command in an unavailability script.

- `Presentation` starts the unavailability proof with a blank part
- `Reducible` proves that a part is successful by reducibility.
- `Pcase L1_2: s[3] <= 7` splits a proof that a part is successful into two cases, depending on whether the third immediate neighbour of the central face has at most or more than 7 sides.
- `Similar to L1_1[4]*` proves that a part is successful by checking that it is contained in the part that was shown to be successful in the left branch of the above `Pcase:` command, reflected and rotated four steps.

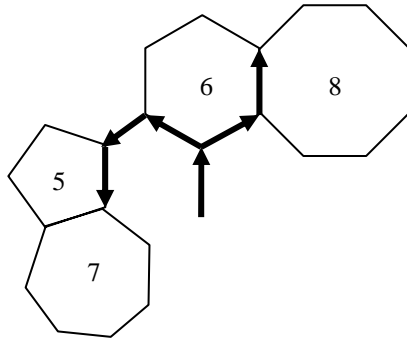
1.3.6 Questions

Questions are binary traversal trees used to efficiently check whether a given reducible configuration occurs in a part. There are no `question` literals in the proof as all questions are compiled from the configuration construction programs. However, it

was very useful to have a readable display for questions to debug the unavoidability commands above; for example we have

```
Eval Compute in (pretty
  (QaskLR Qa6 (QaskLL Qa5 (Qask1 Qa7)) (QaskL Qa8 Qask0))).
= ([6 -L->-L->5-L->7 -R->8-L->*]) : (pretty_printed question)
```

The corresponding traversal checks that the face ahead of the starting edge is a hexagon, then moves left twice, checks for a pentagon ahead, goes left again to check for a heptagon; returning to the starting edge, it then goes right to check for an octagon, then left and returns.



2 The proof shell

The tactic scripts in the proof make a pervasive use of a set of notations defined in `tacticext.v` and `boolprop.v`. As a consequence, their “look and feel” is quite different from standard Coq scripts; for example the usually ubiquitous `Intros` tactic is entirely absent (it is subsumed by the extended `Move` command described below). Altogether, this set of notations define a “shell” that helps insulate the proof script from technicalities of the Coq tactic interpreter, and greatly increases the density and maintainability of the proof script. In particular, several features described below combine to make scripts more reliable: many of Coq’s primitive tactics (e.g., `Intros`, `NewInduction`, or `Inversion`) attempt to perform complex operations with minimal user input, by second-guessing the intended use of the operation; while such tactics are convenient for purely interactive use, they also make proof scripts more fragile. In contrast, our scripting language is designed to make it easier to explicitly specify each operation in detail: in short, the basic steps of these scripts are more like imperative *commands* than speculative tactics.

Indeed, the notations imitate the design of common system command shells: we provide a limited number of commands, but each command takes a set of arguments and switches, so the same command can perform a large range of related operations. The design is quite orthogonal: most commands share the same conventions for arguments and switches, even though the implementation of the corresponding functions in terms of primitive Coq tactics may be completely unrelated.

The commands of our proof shell can be divided amongst four classes: forward and backward chaining commands, which work out the proof from the assumptions or the goal, respectively; bookkeeping commands, which move propositions and data between the assumptions and the goal (possibly after some decomposition), and rewriting commands, which perform equational manipulations.

2.1 Forward chaining

Because forward chaining essentially amounts to adding new assumptions to the proof context, it is closely associated with the declarative style of proof: if each such added assumption is explicitly specified, then a script consisting solely of such forward chaining commands will read very much like a (very detailed) text book proof: a graded list of facts, each backed by a succinct justification. Our main forward chaining command, `Step`, is designed precisely to support this paradigm.

However, it is sometimes more efficient to simply give the combination of lemmas and assumptions that proves a fact; we provide a different command, `Def`, for this usage. The `Def` command is also used to create abstract definitions with explicit equations.

2.1.1 The `Step` command

The general form of this declarative forward chaining command is

```
Step names: proposition [By command-sequence]
```

For example:

```
Step Hxd: (codom embd x) By Rewrite codom_embd; Apply: (hasPn Hrcc).
```

The main difference with Coq’s primitive `Assert` is the `By` part that allows to include a short sequence of commands as “justification” for the new fact. This justification can be omitted (as indicated by the `[]` brackets above), in which case the *proposition* becomes the next subgoal, to which the following commands apply. The last of these is usually a separate `By` command, e.g., the example above is equivalent to:

```
Step Hxd: (codom embd x).
  Rewrite: codom_embd.
  By Apply: (hasPn Hrcc).
```

Another difference with `Assert` is that *names* can be an arbitrary “intro pattern” rather than a simple identifier; this is typically used to decompose an inductive proposition, such as a conjunction:

```
Step [Hyz Hzx]: `y < z` /\ `z < x`.
```

This facility is most often used to decompose an existential fact to introduce a new constant with specific properties:

```
Step [s' Ls' Ds']: (EX s' | (size s') = nc & s' =1 [i](ltn i nc)).
```

Another common use is to explicitly replace a term by an equal:

```
Step []: (k (edge x)) = (h' gc).
```

replaces `(h' gc)` with `(k (edge x))` in the goal, but, unlike the primitive `Replace` tactic, puts the equation subgoal *first*.

2.1.2 The `Done` and `By` commands

Whether used inside a `Step` command or standalone, a `By commands` command is equivalent to `commands; Done` where the `Done` command checks that the remaining goal can be solved by one of the following “trivial” methods:

- `Trivial` (exact or simple instance of an assumption or hint), possibly after `Symmetry`
- `Contradiction` (False assumption)
- `Discriminate` (equality between incompatible terms, after `Intros`)

The standalone `Done` command is the standard justification for trivial `Lemmas` or `Steps`.

Most commands in our scripts are linear: they leave exactly one goal to justify. The vast majority of the other commands are binary (this is due in part to the pervasiveness of binary inductives such as booleans, Peano integers, lists, and simple disjunctions). Quite often one of the branches of these binary commands has an easy or even trivial justification; we provide four commands for such cases: `LeftBy`, `RightBy`, `LeftDone`, and `RightDone`. These are always used as part of a command sequence, e.g.,

```
Case Hdz: disconnected; RightBy Case cross_edge.
```

The following are therefore equivalent

```
Step Ha: (closed e a) By Done.
Step Ha: (closed e a); LeftBy Idtac.
Cut ((closed e a) :: Prop); RightDone; Intros Ha.
```

2.1.3 The `Def` command

The general form of the imperative forward chaining `Def` command is

```
Def [eq-name]: name [: type] := [occ] term
```

In its simplest form, e.g.,

```
Def: Hcp := (config_prog_cubic Hcp').
```

the command just constructs a new fact explicitly, much like the `Assert` command; like `Assert`, the `Def` command can also be used to abstract away values:

```
Def: n := (size p).
```

replaces `(size p)` with a new constant `n`. It is for this usage that the special features of the `Def` command prove most useful:

- Internally, the command uses true abstraction rather than `let`-binding, so it can be used to mask computational contents that causes the `Qed` command to diverge.
- The `: type` option give a convenient syntax for specifying the type of *term*.
- The `occ` option (syntax: `{n1 n2 ...}`) is used to specify which occurrences of *term* should be abstracted, as in


```
Def: n := {1 3}(size p).
```
- The `eq-name` option stores the definition of the new constant as an equation:


```
Def Dn: n := (size p).
```

 generates a new fact `Dn : n = (size p)`. This is quite different from


```
Pose n := (size p).
```

 which generates a *definition* `n := (size p)`. It is not possible to generalize or rewrite such a definition; on the other hand, it is automatically expanded by computation, whereas expanding the equation `Dn` requires explicit rewriting.

2.2 Bookkeeping

Since a proof can be broadly defined as a list of facts supporting a statement, theorem proving is commonly reduced to the accumulation of facts.

Accordingly, several proof methodologies advocate the exclusive use of forward chaining commands. This approach, however, does not work well for large fully formal proofs, simply because such proofs involve too many facts. To make such larger proofs tractable it is therefore crucial to prune irrelevant facts.

Fortunately, we have observed that most of the “new” facts and constants that appear in a proof are directly derived from previous ones by simple operations such as

decomposition, instantiation, reflection, or even simply discharging; in most such cases the new data subsumes the old. The corresponding proof steps are therefore essentially *bookkeeping* steps that just shift and reshape facts into a more useful form. Such steps account for over a third of the scripts of the Four Colour Theorem proof; they outnumber forward chaining steps almost ten to one.

All this makes bookkeeping commands the central feature of our proof shell:

- We define a unique bookkeeping command, `Move`, that can carry out all basic bookkeeping steps, mostly subsuming the `Intros`, `Generalize`, `Rename`, `Clear` and `Pattern` tactics.
- We extend the most frequently used tactics (`Case`, `Elim`, `Apply`, `Rewrite`) with the new `Move`'s bookkeeping features.

2.2.1 The `Move` and `Clear` commands

The basic syntax of the extended `Move` command is

```
Move[: name1 name2 ...] [=> name'1 name'2 ...]
```

For example,

```
Move: x => y.
```

which is equivalent to

```
Rename x into y.
```

Both the left hand and right hand sides of the “=>” are optional. If the right hand side is omitted, then the command performs a discharge step, changing facts in the context into assumptions in the goal. If `n : nat` occurs only in `Hn : (leq n (3))` and in the goal `(lt n (4))`, the command

```
Move: n Hn.
```

changes the goal to `(n : nat) (leq n (3)) -> (lt n (4))` and deletes both `Hn` and `n`. Conversely, applying the command

```
Move=> k Hk.
```

to this goal introduces a new constant `k : nat` and a new fact `Hk : (leq k (3))`, i.e., `Move` where the left hand side of “=>” is omitted behaves like the primitive `Intros` tactic.

More generally, the left and right hand side of a `Move` can have any length, so a single `Move` can perform any combination of renaming, introduction and discharging. A *defective* `Move` command

```
Move.
```

exposes the first assumption in the goal, e.g., it changes the goal `~False` to `False->False`.

We define an extension of the `Clear` command that uses the same conventions:

```
Clear: n Hn.
```

erases both `Hn` and `n` (in that order), just as `Move: n Hn` would, but without changing the goal. The defective

```
Clear.
```

command is equivalent to `Intros _`: it exposes and deletes the first assumption from the goal.

2.2.2 Left hand side `Move`: switches

Items on the left hand side of a `Move` (called *d-items*) are not restricted to simple identifiers; a *d-item* can be an arbitrary Gallina term, which is generalized (replaced by a variable) as with the `Generalize` tactic. Hence,

`Move: (addn i j) => k.`

is equivalent to

`Def: k := (addn i j).`

Each item may also start with a switch, which can be

- `{n1 n2 ...}` an *occ* switch that specifies which occurrences of the term should be generalized (as with the `Def` command), e.g.,
`Move: {2}(S n) (ltnSn n) => m.`
introduces a new constant `m : nat` and adds the assumption `(ltn n m)` to the goal.
- `{name1 name2 ...}` a *clear* switch that specifies that facts `name1 name2 ...` should be deleted after the generalization, e.g.,
`Move: {n}(S n) => m.`
replaces `(S n)` by a new constant `m`, and deletes `n`.
- `{}` a *noclear* switch that specifies that the item should not be deleted from the context, even if it is a simple identifier (the *occ* switch implies this); hence
`Move: {}true.`
generalizes `true` but does not try to delete a (non-existent) variable named `true`.

A *clear* switch may also appear after the last item: the facts and variables it lists are deleted first.

In addition, the `Move:` command also supports the equation-generation feature of the `Def:` command:

`Move Em: (S n) => m.`

replaces `(S n)` in the goal by a new constant `m : nat` and adds a new fact `Em : (S n) = m` to the context. It is equivalent to

`Move: {-1}(S n) (refl_equal nat (S n)) => m Em.`

but different from

`Def Dm: m := (S n).`

since the latter generates the fact `Dm : m = (S n)`. Equation generation implies the *noclear* switch:

`Move Dk: k => j.`

does not delete the constant `k`, as this would always cause an error.

2.2.3 Right hand side `Move=>` switches

The items on the right hand side of a `Move` (called *i-items*) are also not restricted to simple identifiers; an *i-item* can be an arbitrary intro pattern, so `Move=>` can be used to decompose inductive data and propositions, just like the `Intros` tactic. Hence,

`Move: Hxy => [Hx Hy].`

replaces a fact `Hxy : (p x) /\ (q y)` with two separate fact `Hx : (p x)` and `Hy : (q y)`.

The last *i-item* can be the symbol `*`; this indicates that all remaining apparent assumptions/variables in the goal should be moved to the context, with default names as with the bare `Intros` command.

Each *i-item* can be followed by a *clear* switch and/or a *simpl* switch, which is one of

- `/=` execute the `Simpl` tactic.

- // execute the `Done` command for all subgoals on which it succeeds (i.e., it executes `Try Done`).
- `//=` execute `Simpl`; `Try Done`.

A `simpl` switch can immediately follow the ‘=>’, but no switches may appear after the final ‘*’.

When both are present, the `clear` switch always precedes the `simpl` switch textually, but it is executed after. This combination can be used to erase unused fields in a record-like structure, for example, if `u : {n : nat | (ltn (0) n) & (leq n (4))}` and the goal is `(ltn (double (proj1_sig u)) (9))`, then the command

```
Move: u => [n H' Hn] {H'}/=.
```

will decompose `u`, simplify the goal (which becomes `(ltn (double n) (9))`), and erase the superfluous fact `H' : (ltn (0) n)`.

The ‘//’ switch is mostly used to eliminate trivial branches generated by disjunctive intro patterns. For example, in a context containing `p : (seq d)` and `Hp : (ltn (2) (size p))`, the command

```
Move: p Hp => [|x [|y [|z p]]] // _.
```

uses assumption `Hp` to replace `p` by `(Seq x y z & p)`.

2.2.4 Views

With the extensive use of boolean reflection, it is quite frequent to need to decompose the logical interpretation of a fact, rather than the fact itself. This can be achieved by a combination of `Move` switches, e.g. if `Hab : (andb a b)`, then

```
Move: {Hab} (andP Hab) => [Ha Hb].
```

replaces `Hab` with the facts `Ha : a` and `Hb : b` in the context. This operation is so common that the command shell has specific syntax for it:

```
Move/andP: Hab => [Ha Hb].
```

The term after the ‘/’ (`andP` here), which is applied to the assumption before it is generalized, is called a *view*. The view feature is orthogonal to other features of `Move`. In particular it can be used when there are no *i-items*, in which case the view is applied to the first variable or assumption of the goal, e.g., if the goal is `(x =d y) -> G`, then

```
Move/eqP=> Dx.
```

introduces the equation `Dx : x = y`, while the defective

```
Move/eqP.
```

simply changes the goal to `(x = y) -> G`. The view does not have to be a reflection lemma: for instance, after the above command, the command

```
Move/esym.
```

changes the goal to `(y = x) -> G` (`esym` is the standard symmetry lemma `sym_equal` with implicit arguments). One can even use ordinary functions as views: the command sequence

```
Case (odd_double_half n); Move/half: n (odd n) => n b.
```

replaces `n` with `(addn b (double n))`, where `b : bool` is a new constant, using the following lemma from `natprop.v`.

```
Lemma odd_double_half : (n : nat)
  (addn (odd n) (double (half n))) = n.
```

When the view is a reflection lemma, its “application” to the discharged term (which must then be a boolean equation) is fairly flexible. If the right-hand side is false, or if the left-hand side is the negation of the reflected boolean, then the result is the negation of the reflected proposition, e.g., whether we have `Hn : (n =d (2)) = false` or `Hn : (negb n =d (2))`

`Move/eqP`: Hn .
 adds the assumption $\sim(n = (2))$ to the goal. If the reflected boolean is the negation of the left-hand side of the inequality, then we have the converse behaviour: if `Hab` : $(orb\ a\ b) = false$, then
`Move/norP`: $Hab \Rightarrow [Ha\ Hb]$.
 replaces `Hab` with `Ha` : $(negb\ a)$ and `Hb` : $(negb\ b)$. For the trivial reflection lemma `idP`, a negation on the left-hand side of the inequality is interpreted as a negated proposition:
`Move/idP`: $Ha \Rightarrow Ha$.
 replaces the assumption `Ha` above with `Ha` : $\sim a$. Using the `idPn` lemma instead avoids this:
`Move/idPn`: $Hb \Rightarrow Hb$.
 leaves `Hb` unchanged.

2.2.5 The `Case` command

Since *i-items* can be arbitrary intro patterns, in principle the `Move` command can be used to perform arbitrary case analysis. However, neither the name of the `Move` command nor its behaviour on special cases are consistent with this usage. We therefore introduce an extended `Case` command that is better suited for this purpose. Its syntax and behaviour is very close to those of the `Move` command:

`Case` */view* *[[eq-name]: d-item⁺]* *[=> i-item⁺]*.

where *view*, *eq-name*, *d-item* and *i-item* have the same meaning as above. Although the `Case` and `Move` commands are often interchangeable, there are two important differences that separate them:

- The `Case` command *always* performs case analysis on an inductive data type or proposition, whereas for the `Move` command case analysis is triggered by the presence of *i-items* that are not variables.
- The `Case` command has support for the analysis of *dependent* inductives, mostly in the form of an additional `'/'` switch.

The first difference is most apparent when there are no *i-items*: if `b` : `bool`, the command

`Case`: `b`.

generates two subgoals, where `b` has been erased and replaced by `true` and `false`, respectively, whereas

`Move`: `b`.

simply discharges `b`. If the inductive type has parameters then these are not introduced, e.g., given `n` : `nat`, the command

`Case`: `n`.

applied to the goal $(leq\ m\ n)$ erases `n` and leaves two subgoals, $(leq\ m\ (0))$ and $(n : nat)(leq\ m\ (S\ n))$. The defective form

`Case`.

decomposes the top assumption of the goal, e.g., it changes $A \wedge B \rightarrow C$ into $A \rightarrow B \rightarrow C$. A common idiom is to use this together with views to rewrite with an equational assumption, e.g., the command

`Case/esym`.

changes $n = (3) \rightarrow (leq\ m\ n)$ into $(leq\ m\ (3))$.

If the `'=>'` and some *i-items* are present then the differences between the `Case` and `Move` commands boil down to the way the first *i-item* is interpreted;

- For the `Move` command the first *i-item* can be a variable, and if it is a proper pattern it should not contain alternatives, except for trivial branches.

- For the `Case` command the first *i-item* cannot be a variable, and usually contains alternatives. Even if it does not, the `Case` command is expected to generate multiple goals, so the corresponding introductions are performed on all branches, e.g.,

```
Case: (leqP m n) => [Hmn].
```

forks two subgoals with context `Hmn : (leq m n)` and `Hmn : (lt n m)`, respectively. Note that replacing `Case` by `Move` here causes an error, because the intro pattern `[Hmn]` expects an inductive with only one case.

Note that it is possible for the first *i-item* of a `Case` to have either more or fewer branches than the inductive definition decomposed by the `Case`:

- The *i-item* can have more branches if the type of term that is decomposed is a (non-dependent) product, as Coq generates additional goals for the assumptions, e.g., if `Hbc : (A -> B) -> B \ / C`, then

```
Case: Hbc => [Ha | Hb | Hc].
```

generates three branches with contexts `Ha : A`, `Hb : B`, and `Hc : C`, respectively; in the first branch the goal is changed to `B`.
- The *i-item* can have fewer branches if is preceded by a `'//'` or `'//='` switch that eliminates some trivial branches. Note that such a switch would not have the same effect for a `Move` command, because it would be executed *before* the case analysis.

The `eq-name` feature is especially useful for the `Case` command, since the generated equation records the inductive constructor corresponding to each branch. The most common use of this feature is to enumerate conditions, e.g., the command

```
Case Hpx: (p x).
```

generates two subgoals, where the boolean expression `(p x)` has been replaced by `true` and `false`, respectively, and where a new fact `Hpx` has been added (`Hpx : (p x) = true`, and `Hpx : (p x) = false`, respectively). Note that in either case `Hpx` has the form expected by the view feature, and in the first subgoal `Hpx : (p x) = true` is convertible to `Hpx : (p x)`. The `eq-name` feature is very useful to analyse and debug generate-and-test style scripts that prove program properties by generating a large set of input patterns and uniformly solving the corresponding subgoals by computation and rewriting, e.g.,

```
Case: et => [|e' et|]; LeftBy Case: s; Case: e => //; Case: b; Case: w.
```

If the above sequence fails, then it's easy enough to replace the line above with

```
Case: et => [|e' et|. Case Ds: s; Case De: e => //; Case Db: b; Case Dw: w
=> [|s' w'|] //=.
```

Then the first subgoal that appears will be the failing one, and the equations `Ds`, `De`, `Db` and `Dw` will indicate pinpoint its branch. When the constructors of the decomposed type have arguments (like `w : (seq gram_symbol)` above) these need to be introduced in order to generate the `eq-name` equation, so there should always be an explicit *i-item* (`[|s' w'|]` above) that assigns names to these arguments. If this *i-item* is omitted the arguments are introduced with default names; this feature should be avoided except for quick debugging runs (it has some uses in complex tactic sequences, however).

The `Case` command should always be preferred to the `Move` command for decomposing dependent inductive datatypes and propositions. First, the `Case` command only erases names after the initial decomposition, so it can erase dependent parameters that are substituted by the decomposition, for example, given `p : (seq d)` the command

```
Case/lastP: p => [|p z].
```

does a tail decomposition of the sequence p : it generates two subgoals, where p has been replaced with $(Seq0\ d)$ and deleted, and with $(add_last\ p\ z)$ where $z : d$ is a new constant, respectively. Using `Move` here would generate an error, because it would attempt to erase p first.

Moreover, a specific `/` switch can be used indicate the *d-item* with a dependent inductive type. All the *d-items* preceding the `/` are interpreted as the dependent parameters of the item immediately following the `/`, and are not actually generalized, but rather selected for substitution, e.g.,

```
Case: {1 3 4}p / (lastP p) => [|p' z].
```

replaces only the second and third occurrences of p in the goal (the occurrence indices take into account the occurrence of p in the type of $(lastP\ p)$). The *eq-name* feature will generate an equation for the first dependent *d-item*, e.g.,

```
Case Dp: p / (lastP p) => [|p' z].
```

adds $Dp : p = (Seq0\ d)$ and $Dp : p = (add_last\ p'\ z)$, respectively, to the context of the two subgoals it generates. In contrast, the *view* feature applies to the dependent *d-item*, e.g., the command above is equivalent to

```
Case/lastP Dp: p / {p} => [|p' z].
```

There must be at least one *d-item* to the left of the `/` switch; this prevents any confusion with the *view* feature. However, the *d-items* to the right of the `/` are optional, and if they are omitted the first assumption provides the dependent type. For example, the command

```
Case/splitPr Dp: p / => [p1 p2].
```

will transform the goal $(p\ y) \rightarrow (path\ click\ x\ p)$ into $(path\ click\ (cat\ p1\ (Adds\ y\ p2)))$ where $p1, p2 : (seq\ d)$ are new constants, while generating the equation $Dp : p = (cat\ p1\ (Adds\ y\ p2))$.

2.2.6 The `Elim` command

As with the `Case` command, we also extend the primitive `Elim` command of Coq with the features and switches of the `Move` command. In fact the `Elim` command has exactly the same syntax as the `Case` command:

```
Elim [/view] [[eq-name]: d-item+] [=> i-item+].
```

including the restriction that the first *i-item* cannot be a variable, and the `/` dependent *d-item* switch. The main difference with the `Case` command is that the `Elim` command performs inductive elimination rather than decomposition; thus the first *i-item* will introduce induction hypotheses for the various subgoals, along with the constructor arguments, e.g, given $n : nat$,

```
Elim: n => [|n Hrec].
```

will start a proof by induction on n (`Hrec` is the most common identifier in the Four Colour Theorem proof scripts because we almost always name the induction hypothesis `Hrec`). The bookkeeping features provide a very concise and precise notation for constructing the induction hypothesis, specifying which assumptions should be included, which variables should be generalized, etc., as in

```
Elim: w (seq0::bitseq) et {b0 Hw} => [|s w Hrec] lb et.
```

Indeed, the exact induction schema quite rarely appears in extension in our proofs. The bookkeeping features also provide a direct notation for generalized induction on integers, using the idiom

```
Elim: {n}(S n) {-2}n (ltnSn n) => // [m Hrec] n Hnm.
```

which adds a new constant $m : nat$ and the facts $Hrec : (n : nat) (ltn\ n\ m) -> G$ and $Hnm : (ltn\ n\ (S\ m))$ to the context, where G is the current goal. Note that

the type of `Hnm` is actually convertible to `(leq n m)`. It's also common to replace the introduction of `Hmn` with the sequence

```
Rewrite: ltnS leq_eqVlt; Case/orP=> //; Move/eqP=> Dn.
```

which introduces the equation `Dn : n = m` instead, this idiom extends readily to generalized induction on the size of data, e.g.,

```
Elim: {p}(S (size p)) {-2}p (ltnSn (size p)) => // [m Hrec] p Hpm.
```

Despite their apparent similarity, the `Case` and `Elim` tactics generate completely different proof terms: the former compiles into dependent pattern-matching, while the latter generates an application of one of the standard induction lemmas generated by the declaration of the inductive type. Because of this, there are three subtle differences in the behaviour of the extended `Case` and `Elim` commands:

- When using `Elim` with a term t with a dependent inductive type, instances of t are not substituted in the goal (this is a feature of the default elimination lemmas).
- The `view` feature is used to give an alternative induction lemma, rather than a function or reflection lemma to be applied before decomposition, e.g., to do reverse induction on `p : (seq d)`, one uses

```
Elim/last_ind: p => [|p y Hrec].
```

- The `eq-name` feature can only be used when the first *d-item* is a variable. Unlike the `Case` command, the `Elim` command always deletes this variable from the context, and then reintroduces it in each subgoal as a shorthand for the term that was substituted for that goal, e.g.,

```
Elim Dp: p => [|x p' Hrec].
```

applied to a goal `(g p)`, generates two subgoals; `(g (Seq0 d))` with the fact `Dp : p = (Seq0 d)` added to the context, and `(g (Adds x p'))` with the facts `Dp : p = (Adds x p')` and `Hrec : (g p')` added to the context; it signals an error if `p` appears outside the goal.. The `eq-name` feature of `Elim` is mainly useful for the branch debugging use that was described above for the `Case` command.

2.2.7 The `Injection` command

The primitive “decomposition” of an equality amounts to just replacing its right-hand with its left-hand side. It is however fairly natural to decompose an equality that relates two similar instances of a constructor of an inductive type into a set of equalities relating their parameters, e.g., to decompose the equality `Dp1 : (Adds x1 p1) = (Seq x2)` into `p1 = (Seq0 d)` and `x1 = x2`. The primitive `Injection` `Ep12` command does this, but in a fairly restricted manner, so we extended this command with most of the bookkeeping features of the `Move` command, as we had done for `Case` and `Elim`:

```
Injection [ : d-item+ ] [ => i-item+ ].
```

There are several caveats, mostly due to the behaviour of the underlying Coq primitive:

- The `view`, `eq-name` and dependent type features are not supported because they are irrelevant.
- The first *d-item* must be an equation, and command requires separate *i-items* for the generated equations, in *reverse* order:

```
Injection: Dp1 => Dp1 Dx1.
```

replaces the fact D_{p1} above with $D_{p1} : p1 = (\text{Seq0 } d)$ and adds the equation $D_{x1} : x1 = x2$.

- The defective form `Injective`. is not supported because it conflicts with a special case of the primitive `Injection` tactic.

2.3 Backward chaining

In traditional Coq scripts backward chaining tactics tend to dominate as they provide the primary means of constructing deductive proofs. Because of the pervasive use of reflection, backward chaining does not dominate in the proof scripts of the Four Colour Theorem; however it does accounts one fifth of the commands. As most of these backward chaining commands are `Apply` commands, and require some auxiliary bookkeeping operations, we define an extended `Apply` that integrates several bookkeeping features of the `Move` command, and works better in the context of reflection proofs. Even with these improvements, using the standard congruence lemmas (`f_equaln`) is difficult, so we provide specialized `Congr` commands for this.

Note that we also frequently use “as is” many of the basic Coq backward chaining commands such as `Split`, `Exists`, `Left`, `Right`, `Constructor`, `Symmetry` and `Transitivity`.

2.3.1 The `Apply`: command

The extended `Apply` command has the syntax

`Apply`: *term* *d-item* * [*=>* *i-item* ⁺].

The first (non-optional) *term* is never deleted from the context. After discharging the other *d-items*, the command attempts to prove the goal by applying *term*; assumptions in the type of *term* that don’t directly match the goal may generate one or more subgoals. The *i-items* are then executed on these subgoals, as in the `Case` and `Elim` commands; as in these commands, the first *i-item* cannot be a variable. For example, in file `cube.v` the command

`Apply`: (`strict_adjunction` (`Sedge` *g*) *Hate*) => // [*x y Hxy* |].

first generates three subgoals for the remaining assumptions of lemma `strict_adjunction`. One of these is dispatched by the ‘//’ switch; for the first of the other two, two variables *x*, *y* : `cmap` and an assumption *Hxy* : (`tsI CTfe x`) = (`tsI CTfe y`) are moved from the goal to the context.

An important feature of the extended `Apply`: is that it is implemented using the primitive `Refine` tactic rather than `Apply`, which makes it considerably more robust, since it tries to match the goal up to the expansion of constants and the evaluation of subterms. Another benefit is the head *term* can contain wildcards ‘?’; we use this to perform “rewriting” in the `real_model` setoid:

`Apply`: (`eqr_trans` (`mulRCA` ? ? ?)).

changes the goal ``x * y / z = 3`` into ``y * x / z = 3``. The head *term* can even be a reflection lemma: the command

`Apply`: `eqP`.

changes the goal `n = (0)` to `n =d (0)`.

There are, however three cases where the primitive `Apply` is preferable:

- When a single dependent parameter needs to be specified, using the `Apply ... with ...` form.
- When only implicit parameters need to be inferred
- When the *term* has more than five explicit parameters or assumptions (this limitation is solely due to the `camlp4/Ltac` implementation of `Apply`).

2.3.2 The `Apply` command

The view feature of the `Apply` command parallels that of the `Move` command: it allows to change a boolean equality goal into an equivalent proposition. Since *d-items* are useless in this case, this form of the view command has the simpler syntax:

`Apply/view[/view] [=> i-item+]`.

For example, the command

`Apply/eqP`.

changes the goal `n =d (0)` to `n = (0)` (we have seen above that the converse transformation can be accomplished with `Apply`). The trivial reflection lemma `idP` is often used in this context to perform a proof by contradiction:

`Apply/idP=> [Hx]`.

changes either of the goals `(p x) = false` or `(negb (p x))` to `False`, while adding the fact `Hx : (P x)` to the context. Similarly,

`Apply/idPn=> [Hnx]`.

changes the goal `(p x)` into `False` and introduces `Hnx : (negb (p x))`.

The double-view form is useful for boolean equality goals whose right-hand side is not `true` or `false`. It always generates two subgoals, which prove that the propositions reflecting the left and right hand sides of the equality are equivalent. For example,

`Apply/idP/idP=> [Hx | Hy]`.

turns a proof of `(p x) = (q y)` into a proof of `(q y)` using `Hx : (p x)` and a proof of `(p x)` using `Hy : (q y)`. If the left or right hand side of the goal is the negation of the reflected proposition, then the double-view `Apply` generates two subgoals that assert that the propositions are complementary:

`Apply/idP/eqP=> [| [Hx Dy]]`.

turns a proof of `(p x) = (negb y =d (0))` into a proof of `(p x) \ / y = (0)`, and `False` with the assumptions `Hx : (p x)` and `Dy : y = (0)`.

2.3.3 The `Congr` commands

Because of the way matching interferes with dependent type parameters, the command `Apply: congr` will generally fail to perform congruence simplification, even on rather simple cases. We therefore provide a more robust alternative in which the function is supplied:

`Congr term`

where *term* is usually just a function name, e.g., the command

`Congr S`.

simplifies the goal `(S n) = (addn (1) m)` into `n = m`. As this example illustrates, the explicit function name makes the command robust with respect to unfolding and partial evaluation. As with `Apply`, the *term* can contain wildcards, so for instance

`Congr size`.

will work even though `size` is here actually shorthand for `(!size ?)`. In addition, the `Congr` command will try to add up to four wildcards to `term`, as well as trying congruence for binary functions; thus the command

```
Congr addn.
```

will simplify the goals $(\text{addn } n \ m) = (\text{addn } n \ p)$ and $(\text{addn } m \ n) = (\text{addn } p \ n)$ to $m = p$, and will generate the two subgoal $m1 = n1$ and $m2 = n2$ when applied to the goal $(\text{addn } m1 \ m2) = (\text{addn } n1 \ n2)$.

We also define special commands `BoolCongr` and `NatCongr` for congruence simplification of equations in `bool` and `nat`. These specialized commands exploit the commutativity and associativity of the `orb`, `andb`, `addb` and `addn` operators. Both commands work best when the boolean or integers expressions are normalized (to the right) with respect to associativity, e.g., of the form $(\text{orb } b_1 \ (\text{orb } b_2 \ (\text{orb } b_3 \ b_4)))$ where b_1, b_2, b_3 are atomic expressions. For boolean expressions, this is easily accomplished using the extended `Rewrite` command described below; for integers there is a specialized `NatNorm` command that handles the conversions between $(S \ n)$ and $(\text{addn } (1) \ n)$.

2.4 Rewriting

The generalized use of reflection implies that most of the intermediate results we need are properties of effectively computable functions. The most efficient means of establishing such results are computation and simplification of expressions involving such functions, i.e., rewriting. We have therefore defined an extended `Rewrite` command that unifies and combines most of the rewriting functionalities. Even with this powerful command (a single extended `Rewrite` can be equivalent to six to ten primitive ones), nearly a third of the commands in the scripts of the Four Colour Theorem are `Rewrite` commands.

2.4.1 The `Rewrite` command

Whereas the primitive `Rewrite` command can only perform a single rewriting operation in the goal or in a single assumption, the extended `Rewrite` can perform an entire series of such operations in any subset of the goal and/or context. Its general syntax is

```
Rewrite: r-step+ [in fact+]
```

If the ‘in’ part is omitted all rewriting takes place in the goal; otherwise it takes place in the named *facts*, and in the goal as well if the *facts* end with a ‘*’. Hence, given `Dn` : $n = (3)$, `Dm` : $m = (2)$ and `Hmn` : $(\text{leq } m \ n)$, then

- `Rewrite`: `Dn` replaces n with (3) in the goal, and is equivalent to `Rewrite Dn`
- `Rewrite`: `Dn Dm in Hmn` replaces m and n with (3) and (2) , respectively, in the type of `Hmn` (this will fail if `Hmn` appears in the goal)
- `Rewrite`: `Dm in Hmn *` replaces m with (2) in both `Hmn` and the goal

There are three basic kinds *r-steps* (rewrite steps):

- Simplifications, which are denoted by a *simpl* switch such as ‘/=’.
- Proper rewrites, denoted by a *term* (most often a variable) whose type is an equality, possibly generic, possibly with side conditions.
- Unfoldings, denoted by a ‘/’ followed by the name of a defined constant.

For exaple, the command

```
Rewrite: /= /setU1 orbC eqd_sym.
```

will change the goal $(\text{Adds } x \ p \ y)$ into $(\text{orb } (p \ y) \ y =d \ x)$:

- the *simpl* switch will partially evaluate the sequence membership predicate (the corecion $\text{mem} : \text{seq} \rightarrow \text{set}$), yielding $(\text{setU1 } x \ p \ y)$,
- then the definition of setU1 is expanded, exposing the *orb* connective and the ‘=d’ (*eqd*) predicate, yielding $(\text{orb } x \ =d \ y \ (p \ y))$,
- the last two steps use generic rewrite rules to exchange the arguments of *orb* and ‘=d’.

Rewrite and unfolding steps may be inverted by preceding them with a ‘-’ switch: rewrites are done right-to-left and definitions are folded (the latter actually calls a different Coq tactic, *Fold*). For example

`Rewrite: /setU1 -orbA orbCA -/(setU1 x p y).`
 changes $(\text{orb } (\text{setU1 } x \ q \ y) \ (p \ y))$ into $(\text{orb } (q \ y) \ (\text{setU1 } x \ p \ y))$.

The extended *Rewrite* incorporates some of the bookkeeping features of the *Move* command: some steps may start with *r-occ* or *r-clear* switches. More specifically:

- An *r-clear* switch has the form $\sim[\text{clear}]$ where the optional clear is a clear switch $\{name^+\}$. It specifies that the name of the rewrite rule or defined constant should be erased after the step, along with the names appearing in the *clear* switch. For example, given $p := n$ and D_n and D_m as above, the command

`Rewrite: ~/p ~{n}Dn ~Dm.`

replaces p and n with (3) and m with (2) in the goal, and removes p , n , D_n and D_m from the context. A final *r-clear* switch may also follow the last rewrite step. The *clear* part is mandatory for *simpl* steps and the final *r-clear*, but is not allowed on unfolding steps; the *r-clear* switch is not allowed on folding (‘-/’) steps.

- An *r-occ* switch has the form $\{[n_1 \dots] \text{term}\}$. It specifies the exact subterm that will be rewritten by the step. The optional integers $n_1 \dots$ can be used to specify exactly which occurrences of *term* should be rewritten. The *r-occ* switch is only allowed for rewriting and folding steps; it is replaced by a simple *occ* switch for unfolding steps, since the *term* part would be redundant in that case. Given p , D_n , and D_m as above, the command

`Rewrite: {-2}/p {2 3 m}Dm.`

unfolds all but the second occurrence of p , and replaces the second and third occurrences of m with (2). The command

`Rewrite: {y = d x}eqd_sym -{(orb x =d y (p y))}/(Adds x p y).`

changes the goal $(\text{andb } y =d \ z \ (\text{orb } y =d \ x \ (p \ y)))$ into $(\text{andb } y =d \ z \ (\text{Adds } x \ p \ y))$.

Finally, proper rewrite steps may be preceded by a *multiplier*, which specifies repetitions of the step. There are four kinds of multipliers:

- $n!$ the step is repeated exactly n times (n is a positive integer)
- $!$ the step is repeated as many times as possible, and done at least once
- $?$ the step is repeated as many times as possible, optionally
- $n?$ the step is repeated up to n times, optionally

For example, the command

`Rewrite: -?orbA 1?andbC !addn0.`

normalizes all boolean disjunctions with respect to associativity, exchanges the operands of the first conjunction (if there is one), and simplifies all sums of the form $(\text{addn } \dots (0))$ (there should be at least one).

A rewriting step that uses a conditional equation will produce additional subgoals for the conditions, e.g., rewriting `(addn p (subn (3) p))` with

```
Lemma leq_add_sub : (m, n : nat)
  (leq m n) -> (addn m (subn n m)) = n.
```

will generate the subgoal `(leq p (3))`. In such cases the remaining steps are uniformly applied to all such goals. It is quite common to use the ‘?’ multiplier to specify steps that will only apply to auxiliary goals, and a ‘//’ simplification step to dispose of auxiliary goals before returning to the main rewriting sequence.

2.4.2 Locking down subterms

As program proofs tend to generate large goals, it is important to be able to control the partial evaluation performed by the ‘/=’ switch and the location of rewrite steps without having to repeat large subterms of the goal in the proof script. We do this by “clamping down” selected function symbol in the goal, which prevents them from being considered in simplification or rewriting steps. This clamping is accomplished by using the *r-occ* switch together with

```
Lemma lock : (A : Type; x : A) x == (locked x).
```

For example, the command

```
Rewrite: {2 Add}lock /= -lock.
```

changes `(has a (Adds x (Adds y p)))` into `(orb (a x) (has a (Adds y p)))`. Similarly,

```
Rewrite: {-3 orb}lock orbC -lock.
```

exchanges the arguments of the third disjunction in the goal by clamping down all other occurrences of ‘orb’.

It is sometimes desirable to globally prevent a definition from being expanded by simplification; this is done by adding ‘locked’ in the definition, e.g.,

```
Definition cube := (locked (Hypermap cube_moni3)).
```

We provide a special tactic `Unlock` for unfolding such definitions while removing ‘locked’, e.g., the command

```
Unlock cube.
```

replaces `cube` with `(Hypermap cube_moni3)` in the goal.

We found that it was usually preferable to prevent the expansion of arithmetic operations by the partial evaluation switch ‘/=’, unless this allowed the evaluation of a condition. We were able to define a special ‘nosimpl’ form that encapsulates exactly this behaviour, e.g., with

```
Definition addn := (nosimpl plus).
```

the operation `addn` behaves exactly like `plus`, except that `(addn (S n) m)` will not simplify spontaneously to `(S (addn n m))` (the two terms, however, are interconvertible). In addition, the unfolding step `/addn` will replace `addn` directly with `plus`, so the `nosimpl` form is essentially invisible.

2.4.3 Rewriting real expressions

Since our construction of the reals yields a quotient structure, the arithmetic equality between reals cannot be the Leibnitz equality, and we therefore need to use the setoid rewriting features of the `Rewrite` tactic for the reals. Since we prove the Four Colour Theorem for a generic model of the reals, we define a generic real setoid; unfortunately, Coq v7 does not support rewriting with such setoids. We work around

this limitation by defining, in each file that needs to perform such rewriting: on a `real_model R`:

- A specific instance `RR` of the generic real setoid
- An explicit coercion `isR : R -> RR`.
- Specific instances `eqR`, `leqR`, `addR`, ... of the generic real predicates and operations, with the respective morphism lemmas
- Conversions `eqRI`, ... that rewrite generic operations and predicates to specific ones with explicit coercions
- An equation conversion lemma `rwR` that turns a generic equation into a specific one.

This makes rewriting feasible, if somewhat unwieldy: one uses the conversion equations to specialize all predicates and operations above the expression to rewrite, so that they become recognized as morphisms before rewriting with the converted instance of a generic rule. For example, the proof of

`Lemma addrCA : (x1, y, z : R) `x + (y + z) = y + (x + z)`.`

starts by rewriting twice with associativity:

`Move=> x y z; Rewrite: eqRI (rwR (addrA x y z)) (rwR (addrA y x z)).`

This leaves the goal `(eqR (isR `x + y + z`) (isR `y + x + z`))`. Next we need to apply commutativity to the subterm ``x + y``, hence to flag the outer `+` as a morphism, using the command

`Rewrite: addRI.`

to produce the goal `(eqR (isR (addR (isR `x + y`) (isR z))) (isR `y + x + z`))`.

The actual command that completes the proof of `addrCA` is

`By Rewrite: addRI (rwR (addrC x y)) -addRI.`

2.5 Synopsis

We use the following notations for this recapitulation of the syntax of our proof command language:

<code>[a b c]</code>	optional <code>a b c</code>
<code>a / b c</code>	an <code>a</code> , or <code>b c</code>
<code>a b c</code>	an <code>a</code> followed by the symbol <code> </code> followed by <code>b c</code>
<code>a⁺</code>	one or more <code>a</code>
<code>a[*]</code>	zero or more <code>a</code> , i.e., <code>[a]⁺</code> or <code>[a][*]</code>
<code>name</code>	an identifier, e.g. <code>x</code> , <code>Hrec</code> , or <code>path_cat</code>
<code>num</code>	a positive integer, e.g., <code>3</code> or <code>14</code>
<code>int</code>	a signed integer, i.e., <code>[-]num</code>
<code>term</code>	a Gallina term, e.g., <code>(Hrec ? x)</code> or <code>`x + y`</code>
<code>commands</code>	a sequence of proof commands, e.g., <code>Apply: eqP; Rewrite: eqd_sym</code>

2.5.1 Switches, items, and steps

The syntax of commands is built out of the following elements

<code>simpl</code>	<code>/= // // =</code>	simplification switch
<code>occ</code>	<code>{int⁺}</code>	occurrence switch
<code>clear</code>	<code>{name⁺}</code>	clear switch
<code>noclear</code>	<code>{}</code>	“no clear” switch
<code>r-occ</code>	<code>{int[*] term}</code>	rewrite occurrence switch
<code>r-clear</code>	<code>~[clear]</code>	rewrite clear switch

<i>view</i>	<i>/term</i>	view specification
<i>d-item</i>	<i>[occ clear noclear] term</i>	discharge item
<i>d-items</i>	<i>[name]: d-item+</i>	discharge item sequence
<i>dd-items</i>	<i>d-items [/ d-item*]</i>	dependent discharge item sequence
<i>i-pattern</i>	<i>name/ _/[[i-pattern/]*]</i>	intro pattern
<i>i-item</i>	<i>i-pattern [clear] [simpl]</i>	intro item
<i>i-items</i>	<i>=> [simpl][* i-item* [*]]</i>	intro item sequence
<i>mult</i>	<i>[num][!/?]</i>	rewrite rule multiplier
<i>r-step</i>	<i>[-]mult[r-occ/r-clear]term</i>	proper rewrite step
<i>s-step</i>	<i>[~clear]simpl</i>	simplify rewrite step
<i>u-step</i>	<i>[~ occ]/name</i>	unfold rewrite step
<i>f-step</i>	<i>-[r-occ]/term</i>	fold rewrite step
<i>r-item</i>	<i>r-step/s-step/u-step/f-step</i>	rewrite item

2.5.2 Commands

The following are all simple commands:

Step <i>i-pattern: term [By commands]</i>	declarative forward chaining
Def <i>[name]: term [: term] := [occ]term</i>	explicit definition
By <i>commands</i>	complete goal
Done	trivial goal
Clear <i>: name⁺</i>	clear context
Move <i>[view][d-items][clear][i-items]</i>	context ↔ goal moves
Case <i>[view][dd-items][clear][i-items]</i>	inductive decomposition
Elim <i>[view][dd-items][clear][i-items]</i>	inductive recursion
Injection <i>[d-items][clear][i-items]</i>	constructor equation decomposition
Apply <i>: term d-item* [clear][i-items]</i>	backward chaining
Apply <i>view[view][clear][i-items]</i>	reverse reflection
Congr <i>term</i>	congruence simplification
BoolCongr	boolean congruence simplification
NatCongr	(non-negative) integer addition
congruence simplification	
NatNorm	(non-negative) integer addition
normalization	
Rewrite <i>: r-item⁺ [~clear]</i>	rewrite
Unlock <i>name*</i>	unfold ‘locked’ definitions

The following are command sequences

<i>commands₁; LeftBy commands₂</i>	complete 1 st subgoal generated by <i>commands₁</i>
<i>commands₁; RightBy commands₂</i>	complete 2 nd subgoal generated by <i>commands₁</i>
<i>commands; LeftDone</i>	1 st subgoal generated by <i>commands</i> is trivial
<i>commands; RightDone</i>	2 nd subgoal generated by <i>commands</i> is trivial