AUTO2 Documentation (draft)

Bohua Zhan

July 1, 2015

Contents

1 Introduction

AUTO2 is a theorem prover for classical logic written in Isabelle/ML, using Isabelle/HOL as the logical foundation. It is designed to prove theorems or verify steps in a proof encountered in the usual developments of mathematics and computer science. The overall proof strategy is a best-first search through propositions that can be derived from a given list of assumptions, with the goal of eventually deriving a contradiction. The theorem to be proved is always first transformed into this form (for example, $[A, B] \Longrightarrow C$ is first transformed into $[A, B, \neg C] \Longrightarrow \texttt{False}$). In implementing this strategy, emphasis is placed on being able to naturally support rewriting, proof by case analysis, and induction. There is also an internal language of *proof scripts* that allows the user to provide intermediate steps in the proof of a theorem, in a way similar to (but currently independent from) Isar. The prover is packaged as an Isabelle tactic that directly produces a formally verified theorem (one task in the future is to also let it print out steps of the proof, that is both human readable and can be followed through quickly by the computer).

In the first part of this documentation, we give an overview of the algorithm, concluding with some examples from elementary number theory. No knowledge of Isabelle is assumed in this part. In the second part, we describe some implementation details.

2 Overview of algorithm

2.1 Box lattice

The *box lattice* keeps track of which statements are assumed to be true at different places in the proof, and handles the logic of case analysis. A *primitive box* consists of a list of initial assumptions (and possibly introduces initial variables). A general box is a combination of a list of primitive boxes. Each primitive box also *inherits* from a general box, taking all initial variables and assumptions from its parent. Primitive boxes are indexed by integers, so that a general box is specified by a set of integers. The box $\{\}$ is the context with no assumptions. The primitive box 0, inheriting from $\{\}$, contains variables and assumptions for the theorem to be proved. The general box $\{0\}$ (which we call the “home” box) is the initial context. Any other primitive box is expected to inherit, directly or indirectly, from $\{0\}$. By placing a proposition P in a general box i, we claim that P follows from the assumptions (both immediate and inherited) in i. If a contradiction is derived from the assumptions in box i, we say i is *resolved.* The overall goal when proving a theorem is to resolve box $\{0\}$.

The box lattice allows case analysis as follows. Suppose $A \vee B$ is derived in box $\{0\}$, so we may derive a contradiction in $\{0\}$ by deriving contradictions from both A and B. To do so, we create a new primitive box 1 inheriting from $\{0\}$ with A as the additional assumption. Any proposition with derivation depending on A will be placed in box $\{1\}$. If a contradiction is derived in box $\{1\}$ (that is, if $\{1\}$ is resolved), then the proposition $A \Longrightarrow \texttt{False}$, or $\neg A$, can be put into box $\{0\}$. Since $\neg A$ and $A \vee B$ together imply B, we can add B to box $\{0\}$, and begin the second branch of case analysis in $\{0\}$.

For a more complicated example, suppose $A \vee B$ and $C \vee D$ are present in box $\{0\}$, and the proof is to proceed by deriving contradictions in each of the four cases AC, AD, BC, and BD. To do so, we create primitive boxes 1 and 2, both inheriting from $\{0\}$, with A and C respectively as the additional assumption. Once a contradiction is derived from AC, the box $\{1, 2\}$ is resolved. This puts propositions $\neg A$ in box $\{2\}$, and $\neg C$ in box $\{1\}$, from which we can derive B in box $\{2\}$, and D in box $\{1\}$. Showing $AD \Longrightarrow \texttt{False}$ will resolve box $\{1\}$, and showing $BC \Longrightarrow \texttt{False}$ will resolve box $\{2\}$. These will put $\neg A$ and $\neg C$ in box $\{0\}$, which give B and D in box $\{0\}$. Then the last case BD can be checked in box $\{0\}$.

Given two general boxes i and j, we define the *intersection* of i and j to be the general box containing exactly the assumptions in i and j. This is formed by merging the two lists of primitive boxes describing i and j, and removing any redundancies. We say i is the *eq-ancestor* of j, or j is the *eq-descendent* of i, if the intersection of i and j equals j.

2.2 Box items

Each (general) box contains a list of items. There are three types of items: propositions, terms, and implications. A *proposition* is a statement, containing no schematic variables (Isabelle for arbitrary variables), that follows from the assumptions in the box. When a proposition P is placed in a box, it means "P can be proved in this box". Alternatively, since the goal is to derive a contradiction, it also means "proving $\neg P$ resolves this box". A *term* is an expression that appears in the propositions in the box. Following Isabelle notation, we will write a term A as `TERM` A. An implication is a statement of the form $A \Longrightarrow B$ containing schematic variables, that follows from the assumptions in the box.

Each item is assigned an integer called *score*, which directs the best-first search. From the point of view of best-first search, a lower score means it is more attractive to use this item to derive further items. In the current implementation, scores are assigned cumulatively: the initial assumptions in primitive box 0 are given score 0. Any time an item P is derived from a list of items $[Q_1, ..., Q_i]$, the score of P equals the maximum of the scores of Q_i, plus an additional value that depends on P and how P is derived. The scores of initial assumptions in a primitive box other than 0 are calculated similarly from the scores of items that cause this box to be created. In this view, the score also measures the distance of each item from the initial assumptions in primitive box 0.

2.3 Proof steps

Proof steps specify how new items can be derived from old items. The basic principle is "two-item matching". A proof step matches one or two patterns against the same number of items (possibly from different boxes), and produces a new item that logically follows from the matched items, to be placed in the intersection of the boxes containing the matched items (or a descendent box due to rewriting, to be discussed later). Instead of deriving new items, a proof step can also create a new primitive box, or resolve a box (by deriving a contradiction in that box).

We now give some examples of proof steps. Following Isabelle notation, the symbol `?a` denotes a schematic variable (with name "a"), that can be matched to any expression of the right type. When creating the new items, schematic variables in the result pattern are replaced by the corresponding expressions (even if the uninstantiated version of the result also makes sense).

1. (Forward reasoning) If two items match `?m dvd ?n` and `?n dvd ?p`, produce a new item `?m dvd ?p` (divisibility is transitive).

2. (Backward reasoning) If two items match `?k dvd ?m * ?n` and `¬ ?k dvd ?m`, produce a new item `¬ coprime ?k ?n`. This may be difficult to understand as a forward reasoning step. However, since $\neg P$ can be read as a goal to prove P, we can read the above proof step as follows: given `?k dvd ?m * ?n` and needing to prove `?k dvd ?m`, it suffices to show that `?k` and `?n` are coprime.

3. (Simplification rule) If an item matches `TERM` $\texttt{?a}^0$, add new item $\texttt{?a}^0$ `= 1`. This proof step acts as a simplification rule.

4. (Expanding a definition) If an item matches `TERM (prime ?p)`, add new item `prime ?p = (1 < ?p ∧ ∀m. m dvd ?p ⟶ m = 1 ∨ m = ?p)`.

   This proof step acts as expanding the definition of prime. We want to emphasize that, in both this and the previous example, obtaining an equality does not apply this equality to rewrite any of the existing (or future) items. It is only during matching (to be discussed later) where this equality may be used.

5. (Properties of a function) If an item matches `TERM (gcd ?a ?b)`, add new item `gcd ?a ?b dvd ?a`. Sometimes the definition of a function (like gcd) is more naturally expressed as a list of properties. Here when gcd is encountered, we introduce part of the usual definition for gcd.

6. (Resolving a box) If an item matches `?n < ?n`, resolve the box containing that item.

7. (Creating a primitive box) If an item matches `?A ∨ ?B`, create a new primitive box, inheriting from the box containing `?A ∨ ?B`, with the additional assumption `?A`. This supports case analysis as described in Section 2.1.

8. (Adding an implication) If an item matches $\forall$`x. ?A(x)` $\longrightarrow$ `?B(x)`, add new item with implication `?A(?x)` $\Longrightarrow$ `?B(?x);`. For example, given the item $\forall$`m. m dvd p` $\longrightarrow$ `m = 1` $\vee$ `m = p`, add a new implication `?m dvd p` $\Longrightarrow$ `?m = 1` $\vee$ `?m = p`.

9. (Resolving an implication) For any implication `A` $\Longrightarrow$ `B` and proposition `C` such that `A` (considered as a pattern) matches `C`, produce a new item whose proposition is the substituted version of `B`. For example, with implication `?m dvd p` $\Longrightarrow$ `?m = 1` $\vee$ `?m = p`, and item `n dvd p`, produce new item `n = 1` $\vee$ `n = p`.

10. (Adding implication as a backward reasoning step) If an item matches $\neg$`(`$\exists$`x. ?A(x)` $\wedge$ `?B(x))`, add new item `?A(?x)` $\Longrightarrow$ $\neg$`?B(?x)`. This can be read as a backward reasoning step as follows: if we want to show $\exists$`x. A(x)` $\wedge$ `B(x)`, and there is a term x satisfying predicate A, it suffices to show that x also satisfy predicate B.

11. (Arithmetic) Given item n_1 `+ a` $\leq$ n_2, where $n_1 \leq n_2$ are constant natural numbers, produce new item `a` $\leq$ $n_2 - n_1$, where the arithmetic is performed. For example, with `3 + a` $\leq$ `7`, obtain `a` $\leq$ `4`.

Hence new propositions and implications can be added using proof steps. New terms can also be added, although this is usually unnecessary, since all terms appearing in propositions are added internally to the most general box where they appear.

In most cases, a proof step is justified by a single theorem and simply applies that theorem. The additional information that a proof step provides is how that theorem is to be used: whether in the forward or backward direction, and whether there are restrictions on the values of arbitrary variables. In this sense, proof steps can be considered as information on how to use the available theorems to reason within a particular mathematical theory. Ideally, a proof step based on a single theorem is added right after the theorem is proved, and will be available in all subsequent proofs. This means no further "hints" on using the theorem needs to be given to AUTO2 at each proof. For this to be viable, care needs to be taken when writing proof steps, in order to avoid redundant or meaningless steps.

On the other hand, proof steps are user-defined functions, and can carry out arbitrary computation. For example, it is entirely reasonable to have a proof step that takes a system of linear equations as input, and produces the solution as output. In general, any computation following a definite algorithm can be written as a single proof step (anything involving searching through different methods, however, is probably best left to AUTO2).

2.4 Rewriting and matching

A *rewrite table* is used to organize the list of available equalities at any given stage of the proof. These can be equalities coming from identities (simplification rules or definitions), or follow from the initial assumptions (relationships between variables). The rewrite table works only with equality statements without schematic variables (same condition as for propositions in a box).

The interface provided by the rewrite table is relatively simple. As input, one can add a new equality to the rewrite table under a particular box (the rewrite table is designed to handle equalities under different boxes). As output, the rewrite table answers two kinds of queries: it finds the simplest known form for a given expression, and it produces all matches of an expression against a pattern, up to rewriting the expression using the list of known equalities. Only first order matching (not unification or higher order matching) is available, although the matcher can "go inside" $\forall$ and $\exists$ statements, so that proof steps in Examples 8 and 10 above work as intended. Using the simplify function, we can also check whether any two expressions are known to be equivalent.

As an example, suppose `prime p = (1 < p` $\wedge$ $\forall$`m. m dvd p` $\longrightarrow$ `m = 1` $\vee$ `m = p)` is added to the rewrite table. Then the term $\neg$ `prime p` matches the pattern $\neg$`(?A` $\wedge$ `?B)`, with `?A := 1 < p` and `?B :=` $\forall$`m. m dvd p` $\longrightarrow$ `m = 1` $\vee$ `m = p`. This allows de Morgan's law to be applied to $\neg$ `prime p`. The ability to match up to equivalence means we do not need to "choose" whether to rewrite a proposition using a particular equality. Instead, all possible forms of the proposition are available for future derivations.

Since each equality resides in a box, the produced matches (and simplifications) are also conditional on a box. For example, if `a = b` is known in box $\{1\}$, and `a = c` is known in box $\{2\}$, then the expression `a + (b * c)` matches the pattern `?a + (?a * ?a)` in box $\{1, 2\}$, with instantiation `?a := a`. In proof steps, the box in which new items are added is the intersection of the box that the matching is conditional on, and the boxes containing the items.

The matching function also implicitly recognizes associative-commutative functions with units, once they are registered using the requisite theorems. For example, if `x = k * p` is known, then the expression `x * y` matches the pattern `p * ?a`, with `?a := k * y`.

2.5 The main loop

We are now ready to describe the overall algorithm. The state of the proof consists of a box lattice, a list of items in each nonempty box, a rewrite table containing the known equalities, and a priority queue containing future updates. Each update (except the initial one) comes from a proof step, and contains one of three types of actions: adding items to a box, creating a primitive box, and resolving a box. The priority of an update is the score of the items to be added (smaller value means higher priority). The initial state consists of an empty box lattice, and one update in the queue, which creates the primitive box 0 containing the variables and assumptions of the theorem to be proved. Each "step" in the proof pulls the update with the lowest score from the queue, checks whether the action to be taken is redundant (whether the box it updates is already resolved, and if not, whether the items / boxes it intends to add are already there up to equivalence). If the action is non-redundant, it is applied to the state. For each proposition added in the update, new updates are created to add any new terms it contains to the same box (thus adding new terms is not done immediately, but at future iterations). In addition, for each new item added to the state, the following actions are performed:

1. The new item is matched against all proof steps. For proof steps that match two items, it is matched while paired with each of the other items in the state. All new updates created are added to the priority queue.

2. If the new item is an equality, it is added to the rewrite table. Then every item and every pairs of items are matched against all proof steps, checking to see whether the new equality will introduce new matches. All new updates (those that require the new equality to be created) are added to the priority queue. Of course, optimizations can be made in an actual implementation. For example, only items containing either the left or right side of the equality, up to equivalence, need to be matched. However, this procedure, which we call *incremental matching*, still takes up most of the running time, and one aim in the future is to find ways to reduce repetitive work here.

This procedure continues, until box $\{0\}$ is resolved, or until certain timeout conditions are reached. One way to measure the amount of work done is by counting the number of updates pulled from the priority queue (including those not applied due to redundancy). We call this the number of "steps" used. One can then tell the program to give up after reaching a certain number of steps.

2.6 Normalization (experimental)

When an item is first added via an update, we perform certain transformations to put it in "normal" form. This results in one or more items that together is equivalent to the original item. Some of the normalizations we currently perform are:

- For item of form $A_1 \wedge \cdots \wedge A_n$, split into items $A_1, \ldots, A_n$.
- For item of form $\neg(A_1 \vee \cdots \vee A_n)$, split into items $\neg A_1, \ldots, \neg A_n$.
- Replace $A = \texttt{True}$ and $A \neq \texttt{False}$ by A. Replace $A = \texttt{False}$ and $A \neq \texttt{True}$ by $\neg A$.
- For associative-commutative operations, put in normal form (remove unnecessary parentheses and sort arguments).
- Sort quantifiers of the same type to a standard order. That is, $\forall x\ y.A(x, y)$ and $\forall y\ x.A(x, y)$ should have the same normalization.

We also use normalization to perform some standard simplifications. However, one must be aware that it can cause the program to miss certain matches, as the equalities used for the simplications are not added to the rewrite table.

2.7 Mutable states (experimental)

Mutable states is an experimental feature that slightly modifies the main loop. They are essentially items that can "change state", which is immediately reflected in subsequent matches (rather than performed in a future update). For example, suppose a proposition $A_1 \vee A_2 \vee \cdots \vee A_n$ is added to the state, which already contains $\neg A_1$ and $\neg A_2$. Without mutable states, the best we can do is to produce updates adding propositions $A_2 \vee A_3 \vee \cdots \vee A_n$ and $A_1 \vee A_3 \vee \cdots \vee A_n$, and then an update adding $A_3 \vee \cdots \vee A_n$ after one of those is applied. With large disjunctions this can become wasteful. Instead, we store a disjunction such as $A_1 \vee \cdots \vee A_n$ as a *mutable state*. When matched against $\neg A_1$, it "changes state" to $A_2 \vee \cdots \vee A_n$. All subsequent matches are done on the new state. For example, it can then be matched against $\neg A_2$ and change state to $A_3 \vee \cdots \vee A_n$.

In fact we currently allow two kinds of mutable states coming from disjunctions: the *active* state and the *passive* state. An active state, generated from propositions of the form $A_1 \vee \cdots \vee A_n$, will attempt to resolve the current box by case checking. That is, it will create a new box with assumption A_1, and when resolved A_2, and so on. A passive state, generated from propositions of the form $A_1 \longrightarrow \cdots \longrightarrow A_{n-1} \longrightarrow A_n$, will not attempt case checking, but simply matches against propositions $A_1, \ldots, A_{n-1}, \neg A_n$ which will simplifies it, until one of $\neg A_1, \ldots, \neg A_{n-1}, A_n$ can be derived.

2.8 Retro-handlers and induction (experimental)

Sometimes it can be convenient to add propositions that do not logically follow from the assumptions, but can be assumed for the purpose of the proof, because any proof of contradiction using it can be transformed to one that does not. The most immediate example is using an existence fact. If `∃x. P x` is derived in a certain box, then we can add a new proposition `P x` to the same box, where `x` is a variable that did not appear anywhere else in the proof. While `P x` does not logically follow from the assumptions (which do not even mention `x`), any proof using `P x` and with a conclusion that does not involve `x` can be transformed into a proof that does not use `P x`, by applying the theorem `∃x. ?P x ⟹ (⋀x. (?P x ⟹ ?Q)) ⟹ ?Q` (`exE` in Isabelle). We allow propositions such as `P x` to be added to boxes. When a box is resolved in a way that uses `P x`, a *retro-handler* is invoked to retroactively remove the dependence.

Another way to support using an existence fact is to create a new primitive box with variable `x` and initial assumption `P x`. When it is resolved, the proposition `⋀x. (P x ⟹ False)` will be available to the parent box, so that a contradiction can be derived in the parent box by applying `exE`. The approach currently used, involving retro-handlers, is chosen mainly to reduce the number of primitive boxes needed in a proof.

A second use of retro-handlers is to support induction. In particular, we support the use a particular form of induction rule using retro-handlers. The form of the theorem is:

$$[P(o), \forall t. P'(t) \longrightarrow P(t)] \Longrightarrow P(t),$$

where o is the value corresponding to the base case of the induction, and $P'(t)$ denotes the previous case of the induction. Some examples are:

- For natural numbers: $[P(0), \forall n.P(n-1) \longrightarrow P(n)] \Longrightarrow P(n)$.
- For lists: $[P([]), \forall l.P(\mathit{tail}\ l) \longrightarrow P(l)] \Longrightarrow P(l)$.
- For trees: $[P(\mathit{leaf}), \forall t.P(\mathit{lsub}\ t) \wedge P(\mathit{rsub}\ t) \longrightarrow P(t)] \Longrightarrow P(t)$.

We can apply an induction rule of this form by adding an assumption to the box corresponding to the previous case (more general induction rules are implemented using box callbacks, see Section 2.9). We give the detailed procedure for the case of natural numbers.

To perform induction on a natural number n, first a primitive box is created with assumption $n = 0$. Induction begins when this box is resolved adding $n \neq 0$ to the parent box. Let $[P_1(n), ..., P_i(n)]$ be the list of initial assumptions involving n in the box containing $n \neq 0$, then the statement on which we want to apply the induction principle is $P(n)$: $[P_1(n), ..., P_i(n)] \Longrightarrow$ `False`. We add $P(n-1)$ to the box containing $n \neq 0$. Resolving that box in a way that depends on $P(n-1)$ means $P(n-1) \Longrightarrow P(n)$ is proved. From $n \neq 0$ we may obtain $P(0)$. Together they imply $P(n)$ for all values of n using the induction rule. So the dependence on $P(n-1)$ can be removed from the derived contradiction. To perform induction with other variables held arbitrary, we generalize over these variables in the statement of P.

In this approach, starting an induction does not impose much overhead − it simply adds one more proposition to the state. Checking the $n = 0$ case is also shared with a potential proof by case analysis on $n = 0$ and $n \neq 0$.

Strong induction is implemented similarly. The general form of a strong induction rule is:

$$(\bigwedge n.(\forall m < n.P(m)) \Longrightarrow P(n)) \Longrightarrow P(n).$$

Such a rule holds for natural numbers, as well as any other set that is well-ordered, such as the set of finite multisets of natural numbers. To invoke strong induction, we add the assumption $\forall m < n.P(m)$. This assumption can be removed using the strong induction rule when the box is resolved.

2.9 Box callbacks

Box callbacks gives another way to perform some procedure when a box is resolved. The difference between box callbacks and retro-handlers is as follows. Retro-handlers are called during the construction of the theorem resulting from the resolution of a box, removing extra assumptions introduced in the box. Box callbacks are called after the theorem is produced, with the theorem as its input, and outputing an update (which can be creating a new box, with its own box callback).

A basic application of box callbacks is for applying a *Horn clause*, of the form $A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow B$, where each A_i is of the form

$$\bigwedge x_{i1} \cdots x_{ik}.\ A_{i1} \Longrightarrow \cdots \Longrightarrow A_{im} \Longrightarrow B_i,$$

where each A_{ij}, B_i and B are expressions in object logic. To apply such a clause (after instantiating any schematic variables) to obtain B, we first add a box corresponding to A_1, with $x_{1\star}$ as variables and $[A_{i\star}, \neg B_i]$ as assumptions. When this box is resolved, a second box is added corresponding to A_2, and so on, until the resolution of the box corresponding to A_n, which allows us to obtain B.

Being able to apply any Horn clause of this format allows us to perform induction on inductively defined propositions and types. See the development of Hoare logic for examples.

2.10 Proof scripts

Proof scripts allows the user to provide intermediate steps in the proof of a theorem to AUTO2. Ideally, only intermediate steps that require some creativity to find need to be provided, while the routine parts between them can be filled in automatically by the program. We currently allow the following commands in the proof scripts.

- `CASE` t. This command adds a new box with assumption t, and sets the focus to resolving this box (the only thing affected by the location of focus is scoring. See Section 3.5).
- `OBTAIN` t. This command adds a new box with conclusion t, and set the focus to resolving this box (this is exactly the same as `CASE` $\neg t$).
- cmd_1 `THEN` cmd_2. Perform cmd_1 and, when it is finished (the new box is resolved), perform cmd_2.
- cmd_1 `WITH` cmd_2. The command cmd_1 creates a new box (not one of the induction commands below). Use cmd_2 when resolving that box.
- `CHOOSE` $(x, P(x))$. Here x is a string denoting the name of a fresh variable, and P is a proposition involving x. This command adds a new box with conclusion $\exists x.P(x)$. When this box is resolved, the existence statement is instantiated (via a box callback) with variable x. Likewise, `CHOOSE` $(x, y, P(x, y))$ will try to instantiate variables x, y satisfying P, etc. The command `CHOOSES` $[s_1, \ldots, s_n]$ is equivalent to `CHOOSE` s_1 `THEN` $\cdots$ `THEN` `CHOOSE` s_n.
- `INDUCT` (n, `[OnFact` $n \neq o$`, Arbitrary` m`])`. Apply induction rule as described in Section 2.8. Here o is the value of the base case of the induction. Wait for proposition $n \neq o$ to become available, then add the

induction assumption for n, holding m to be arbitrary. The latter is omitted if no variable is held arbitrary, and `Arbitraries` is used if multiple variables are held arbitrary. This and all induction commands below require the proper induction rule to be registered.

- `STRONG_INDUCT` $(n, [\cdots])$, `VAR_INDUCT` $(n, [\cdots])$. Perform strong induction (Section 2.8) or general variable induction (Section 2.9). Arbitrary variables can be specified.
- `PROP_INDUCT` $(exp, [\cdots])$. With *exp* as an initial assumption, perform propositional induction on *exp*. The proposition *exp* should be in the form $f(x_1, \ldots, x_n)$, where f is an inductively defined predicate. The induction is standard if each x_i is a variable. Otherwise a well-known trick for propositional induction is performed, replacing any non-variable x_i with fresh variable v_i, and adding the equality $v_i = x_i$. The extra equalities are removed and replacements undone at the end of induction.

In general, a command in the proof script adds either an item or a box, the resolution of which may require further commands. The command interface should be flexible enough that new commands may be added by the user.

2.11 Examples in number theory

We now give some examples from elementary number theory. These are based on theories `Primes` and `UniqueFactorization` in `HOL/Number_Theory`. There are often multiple paths to proving a theorem, and which path the program takes depends on details of the implementation. We describe just one possible path. Moreover, many items not mentioned here, including ones irrelevant to the proof, will be added during the best-first search. In the first few examples, we will focus on the details of boxes, proof steps and matching, while later on we will skip these details to focus on higher level issues such as interpretation of proof scripts.

2.11.1 Example 1: `prime_odd_nat`

The statement of the theorem is

$$\texttt{prime p} \Longrightarrow \texttt{p > 2} \Longrightarrow \texttt{odd p}.$$

In Isabelle, `odd p` is actually an abbreviation for `¬ even p`, and `even p` is in turn an abbreviation for `2 dvd p`. Thus, the goal is to derive a contradiction from the list of assumptions [`prime p`, `p > 2`, `2 dvd p`].

1. From `TERM (prime p)`, add `prime p = (1 < p ∧ ∀m. m dvd p ⟶ m = 1 ∨ m = p)`.
2. Using the equality from step 1, `prime p` matches pattern `?A ∧ ?B`. From this, we conclude `1 < p` and `∀m. m dvd p ⟶ m = 1 ∨ m = p`.

3. From ∀m. m dvd p ⟶ m = 1 ∨ m = p, add implication ?m dvd p ⟹ ?m = 1 ∨ ?m = p.

4. The implication in step 3 matches the initial assumption 2 dvd p, creating 2 = 1 ∨ 2 = p.

5. The disjunction in step 4 triggers a case analysis, creating primitive box 1 under $\{0\}$ with initial assumption 2 = 1.

6. In box $\{1\}$, 2 = 1 is trivially a contradiction. This resolves box $\{1\}$, adding 2 ≠ 1 to box $\{0\}$.

7. From 2 ≠ 1 and 2 = 1 ∨ 2 = p, conclude 2 = p in box $\{0\}$.

8. The equality 2 = p allows p > 2 to be matched with pattern ?n > ?n, which gives a contradiction, resolving box $\{0\}$.

2.11.2 Example 2: not_prime_eq_prod_nat

The statement of the theorem is

[n > 1, ¬ prime n] ⟹ ∃m k. n = m * k ∧ 1 < m ∧ m < n ∧ 1 < k ∧ k < n.

The goal is to derive a contradiction from the list of assumptions [n > 1, ¬ prime n, ¬ ∃m k. n = m * k ∧ 1 < m ∧ m < n ∧ 1 < k ∧ k < n].

1. From TERM (prime n), add prime n = (1 < n ∧ ∀m. m dvd n ⟶ m = 1 ∨ m = n).

2. ¬ prime n matches ¬ (?A ∧ ?B). Applying de Morgan's law, we get proposition ¬ 1 < n ∨ ¬ ∀m. m dvd n ⟶ m = 1 ∨ m = n, which generates an active mutable state.

3. The mutable state from step 2 matches n > 1 to derive ¬ ∀m. m dvd n ⟶ m = 1 ∨ m = n.

4. TERM (¬ ∀m. m dvd n ⟶ m = 1 ∨ m = n) matches TERM (¬ ∀x. ?A), adding an equality rewriting it to ∃m. ¬ (m dvd n ⟶ m = 1 ∨ m = n).

5. The result of step 4 matches pattern ∃x. ?A. Using the existence fact, we get proposition ¬ (m dvd n ⟶ m = 1 ∨ m = n), with m a fresh variable.

6. From step 5, it suffices to show m dvd n ⟶ m = 1 ∨ m = n. So m dvd n holds and it suffices to show m = 1 ∨ m = n (that is, add ¬ (m = 1 ∨ m = n) as a fact). Applying de Morgan's law to the latter, add m ≠ 1 and m ≠ n as facts.

7. m dvd n (from step 6) rewrites to ∃k. n = m * k. Using the existence statement, add proposition n = m * k, with k a fresh variable.

8. From `n = m * k`, and `n > 0` (which follows from `n > 1`), we get `1` $\leq$ `m` $\leq$ `n` and `1` $\leq$ `k` $\leq$ `n`.

9. Since it suffices to show $\exists$`m k. n = m * k` $\wedge$ `1 < m` $\wedge$ `m < n` $\wedge$ `1 < k` $\wedge$ `k < n`, and `n = m * k`, it suffices to show `1 < m` $\wedge$ `m < n` $\wedge$ `1 < k` $\wedge$ `k < n` (the negation of this proposition is added as a fact).

10. de Morgan's law rewrites $\neg$`(1 < m` $\wedge$ `m < n` $\wedge$ `1 < k` $\wedge$ `k < n)` to $\neg$`(1 < m)` $\vee$ $\neg$`(m < n)` $\vee$ $\neg$`(1 < k)` $\vee$ $\neg$`(k < n)`. This is added as a mutable state. Primitive box 1 is created under $\{0\}$ with assumption $\neg$`(1 < m)`.

11. In box $\{1\}$, it suffices to show `1 < m`, and `m` $\neq$ `1` is known in box $\{0\}$, so it suffices to show `1` $\leq$ `m` in box $\{1\}$. But this is known from step 8. So box $\{1\}$ is resolved. The propositions `1 < m` is added to $\{0\}$ and the mutable state becomes $\neg$`(m < n)` $\vee$ $\neg$`(1 < k)` $\vee$ $\neg$`(k < n)`.

12. Primitive box 2 is created under $\{0\}$ with assumption $\neg$`(m < n)`. This is resolved similarly as before, using `m` $\neq$ `n` from step 6 and `m` $\leq$ `n` from step 8. This adds `m < n` to $\{0\}$ and the mutable state becomes $\neg$`(1 < k)` $\vee$ $\neg$`(k < n)`.

13. Primitive box 3 is created under $\{0\}$ with goal `1 < k`. Since `1` $\leq$ `k` is known from step 8, it suffices to show `1` $\neq$ `k`. Equivalently, we may add `1 = k` as a fact.

14. But from `1 = k` and `n = m * k`, we get `n = m`, contradicting `m` $\neq$ `n` from step 6. This resolves box $\{3\}$, and adds `1 < k` as fact and `k < n` as goal to $\{0\}$.

15. Since `k` $\leq$ `n` is known from step 8, it suffices to show `k` $\neq$ `n`. But `k = n` means `n = m * k` matches the pattern `?n = ?m * ?n`. Together with `n > 0`, we get `m = 1`, which contradicts `m` $\neq$ `1` from step 6. This resolves box $\{0\}$ and proves the theorem.

This example shows how the goal of proving the conjunction of four propositions is transformed via de Morgan's law to checking four cases, which is then done one by one using a mutable state.

2.11.3 Example 3: `prime_power_mult`

In this example, we show how awareness of associative-commutative properties of functions during matching is crucial for proving a more difficult theorem. Since the proof is longer, we omit some details.

The statement of the theorem is

$$\texttt{prime p} \Longrightarrow \texttt{x * y = p}^{k} \Longrightarrow \exists \texttt{i j. x = p}^{i} \wedge \texttt{y = p}^{j}.$$

Since this proof involves induction with arbitrary variables, which is not automatically tried, we need to provide the following proof script:

```
CASE "k = 0" THEN INDUCT ("k", [OnFact "k ≠ 0"] @ Arbitraries ["x", "y"])
```

The proof proceeds as follows:

1. Following the proof script, create primitive box 1 with assumption `k = 0`. This acts as the first step to both simple induction and case analysis on `k = 0` and `k` $\neq$ `0`.

2. Add equality rewriting the goal to $\exists$`i. x = p`^{i} $\wedge$ $\exists$`j. y = p`^{j}.

3. On seeing goal $\exists$`i. x = p`^{i} $\wedge$ $\exists$`j. y = p`^{j}, create primitive box 2 under box $\{0\}$ with goal $\exists$`i. x = p`^{i} (via de Morgan's law and case analysis, similar to Example 2). With this case analysis, box $\{1\}$ can be resolved. We omit the details. Then, `k` $\neq$ `0` is added to box $\{0\}$.

4. On seeing `k` $\neq$ `0`, following the proof script, invoke induction generalizing variables `x` and `y`. This adds proposition $\forall$`x y. x * y = p`$^{k-1}$ $\longrightarrow$ $\exists$`i j. x = p`^{i} $\wedge$ `y = p`^{j} to box $\{0\}$. Using this we add implication `?x * ?y = p`$^{k-1}$ $\Longrightarrow$ $\exists$`i j. ?x = p`^{i} $\wedge$ `?y = p`^{j}.

5. Since `n` $\neq$ `0`, the term `p`^{k} rewrites to `p * p`$^{k-1}$.

6. The items `prime p` and `x * y = p`^{k} match a proof step with patterns `prime ?p` and `?m * ?n = ?p * ?q`. This proof step adds a new item `p dvd x` $\vee$ `p dvd y` (The idea here is to use the theorem `[prime p, p dvd m * n]` $\Longrightarrow$ `p dvd m` $\vee$ `p dvd n`, even if the second condition is hidden as an equality `p * q = m * n`). Primitive box 3 is added under $\{0\}$, with assumption `p dvd x`.

7. In box $\{3\}$, we get `x = ka * p` from `p dvd x`, using a fresh variable `ka`.

8. The items `p > 0` and `x * y = p`^{k} match the patterns `?p > 0` and `?p * ?a = ?p * ?b` (but with `?a = ?b` unknown), when `x * y = p`^{k} is rewritten as `(ka * p) * y = p * p`$^{k-1}$. From this we conclude `ka * y = p`$^{k-1}$ in $\{3\}$. This is the step that really exercises pattern matching with rewriting and with associative-commutative functions.

9. The result of step 8 matches the left side of implication from step 4 (the induction hypothesis), concluding $\exists$`i j. ka = p`^{i} $\wedge$ `y = p`^{j} in $\{3\}$. The result rewrites to $\exists$`i. ka = p`^{i} $\wedge$ $\exists$`j. y = p`^{j}.

10. The two existence statements are used to add propositions `ka = p`^{i} and `y = p`^{j} in $\{3\}$, with `i`, `j` being fresh variables.

11. The term `ka * p` matches `?p * ?p`$^{?i}$, which rewrites it to `p`$^{i+1}$. So `x = ka * p = p`$^{i+1}$ is known in $\{3\}$.

12. The goal $\exists$`i. x = p`^{i} in box $\{2\}$ generates the implication `x = p`$^{?i}$ $\Longrightarrow$ `False`. Since the result of step 11 matches the left side of this implication, box $\{2, 3\}$ is resolved.

13. Resolving box $\{2, 3\}$ adds facts $\exists$`i. x = p`^{i} to box $\{3\}$ and $\neg$ `p dvd x` to box $\{2\}$. Then goal $\exists$`j. y = p`^{j} is added to box $\{3\}$.

14. In box $\{3\}$, goal $\exists$`j. y = p`^{j} generates the implication `y = p`$^{?j}$ $\Longrightarrow$ `False`. Since `y = p`^{j} (from step 10) matches the left side of this implication, box $\{3\}$ is resolved. Facts $\neg$ `p dvd x` and then `p dvd y` are added to the home box $\{0\}$.

15. Checking the case `p dvd y` proceeds in box $\{0\}$ in the same way as checking the case `p dvd x` in box $\{3\}$, again using the case analysis provided by primitive box 2. The rest is omitted.

The pattern of case analysis using primitive boxes 2 and 3 here is essentially as described at the end of Section 2.1.

2.11.4 Example 4: `prime_factor_nat`

The last three examples will focus on proof scripts, and omit most of the details at proof step level. In this example, we prove the fact that any natural number greater than 1 has a prime divisor. The statement of the theorem is

$$\texttt{n} \neq \texttt{1} \Longrightarrow \exists\texttt{p. p dvd n} \wedge \texttt{prime p.}$$

We follow a proof by strong induction on `n`.

Since strong induction is not invoked by default, we need to supply a command for it. The command is `STRONG_INDUCT ("n", [])`. The empty list means there are no arbitrary variables. The command immediately adds the fact $\forall$`m<n. m` $\neq$ `1` $\longrightarrow$ `(`$\exists$`p. prime p` $\wedge$ `p dvd m)` to box $\{0\}$, which introduces the implication `?m < n` $\Longrightarrow$ `?m` $\neq$ `1` $\longrightarrow$ `(`$\exists$`p. prime p` $\wedge$ `p dvd ?m)`. The rest of the proof proceeds as follows:

1. Since `n dvd n`, we obtain the fact $\neg$ `prime n`. This gives $\neg$`(1 < n)` $\vee$ $\neg$`(`$\forall$`m. m dvd n` $\longrightarrow$ `m = 1` $\vee$ `m = n)`.

2. In the first case, we have $\neg$`(1 < n)` or `n` $\leq$ `1`. Since `n` $\neq$ `1`, we get `n < 1` and then `n = 0`. Since `p dvd 0` holds for any `p`, it suffices to show $\exists$`p. prime p`. But this is a result proved earlier.

3. In the second case, instantiate with variable `m` and proposition $\neg$`(m dvd n` $\longrightarrow$ `m = 1` $\vee$ `m = n)`. This becomes a conjunction of facts `m dvd n`, `m` $\neq$ `1`, and `m` $\neq$ `n`. From `m dvd n` we get `m` $\leq$ `n`, which when combined with `m` $\neq$ `n` gives `m < n`. From the strong induction assumption, we get $\exists$`p. prime p` $\wedge$ `p dvd m`. Instantiate with variable `p` and assumptions `prime p` and `p dvd m`. From `prime p` it suffices to show `p dvd n`, but this follows from `p dvd m` and `m dvd n`.

After the proof of this theorem, we add it as a backward proof step. That is, we add a proof step that looks for goals matching the pattern $\exists$`p. p dvd ?n` $\wedge$ `prime p`, and for each match, creates a goal `?n` $\neq$ `1` (with instantiated `?n`). This proof step will be used in the next example.

2.11.5 Example 5: Infinitude of prime numbers

In this section, we show how to formalize Euclid's proof of the infinitude of prime numbers using AUTO2. Since this proof requires substantial creativity, we cannot expect the computer to come up with it without any hints. However, we will see that just a few lines of hints will allow the computer to obtain the proof.

The main lemma is that given any natural number n, there is a prime p greater than n. The statement of the lemma is

$$\exists \texttt{p. prime p} \wedge \texttt{n < p}.$$

We use the following proof script for the lemma:

```
CHOOSE "(p, prime p ∧ p dvd fact n + 1)" THEN
CASE "p ≤ n" WITH OBTAIN "p dvd fact n")
```

The proof proceeds as follows:

1. The first command creates a new box with goal ∃`p. prime p` ∧ `p dvd fact n + 1` (here `fact` denotes the factorial function). The goal matches the conclusion of the theorem proved in Example 4. The backward proof step for that theorem produces new goal `fact n + 1` ≠ `1`. Alternatively, we have fact `fact n + 1 = 1`. This becomes `fact n = 0`, which contradicts the fact `fact n` ≥ `1` (which is added on seeing the term `fact n`). This resolves the new box. The resulting existence statement is instantiated with variable `p` and assumptions `prime p` and `p dvd fact n + 1`.

2. Since we have `prime p`, it suffices to show `n < p`. Following the second line of the command, we add a new box with assumption `p` ≤ `n`, and under that a new box with goal `p dvd fact n`. By backward reasoning with theorem `[1` ≤ `p, p` ≤ `n]` ⟹ `p dvd fact n`, it suffices to show `1` ≤ `p`. But we have `p > 1` from `prime p`. This resolves the goal `p dvd fact n`. From `p dvd fact n` and `p dvd fact n + 1`, we obtain `p dvd 1`, which means `p = 1`, contradicting `p > 1`. This resolves the box with assumption `p` ≤ `n`.

3. The new fact ¬`(p` ≤ `n)` from resolving the previous box shows `n < p`, which is what we want.

The second line of the command can actually be simplified to `OBTAIN "p dvd fact n"`, since the goal of proving `n < p` already provides the assumption `"p` ≤ `n"`. However, writing the proof script this way may be confusing to the human reader.

This lemma is added as a resolve proof step. That is, any box containing a goal matching the pattern ∃`p. prime p` ∧ `?n < p` will be resolved.

Now we come to the main theorem. The statement is

$$\neg\ \texttt{finite}\ \{\texttt{p.\ \ prime p}\}.$$

The proof script has only one line:

```
CHOOSE "(b, prime b ∧ Max {p. prime p} < b)"
```

This command adds a new box with goal `¬(∃b. prime b ∧ Max {p. prime p} < b)`. This goal is immediately resolved due to the proof step added from the previous lemma. The resulting existence fact is instantiated with variable `b`, giving facts `prime b` and `Max {p. prime p} < b`. From the latter fact, we obtain `b ∉ {p. prime p}`, which contradicts `prime b`. This proves the theorem.

2.11.6 Example 6: Unique factorization theorem

In this section, we show the formalization of the unique factorization theorem. Following theory `UniqueFactorization` in the HOL library, we state the theorem in terms of multisets. That is: any natural number greater than zero can be written uniquely as the product of a multiset of prime numbers. This avoids dealing with lists and permutation of lists.

First, we show the existence of factorization. The statement of the theorem is:

$$\texttt{n > 0} \Longrightarrow \exists\texttt{M. } (\forall\texttt{p} \in \texttt{set\_of M. prime p}) \wedge \texttt{n = (}\prod\texttt{i}\in\texttt{\#M. i).}$$

Here `M` takes value in the set of finite multisets of natural numbers. The notation `∏i∈#M. i` means "the product of `i` when `i` ranges over `M`, counting multiplicity".

The proof script is:

```
1 STRONG_INDUCT ("n", []) THEN
2 CASE "n = 1" WITH OBTAIN "n = (∏i∈#{#}. i)" THEN
3 CASE "prime n" WITH OBTAIN "n = (∏i∈#{#n#}. i)" THEN
4 CHOOSES ["(m, k, n = m * k ∧ 1 < m ∧ m < n ∧ 1 < k ∧ k < n)",
5          "(M, (∀p∈set_of M. prime p) ∧ m = (∏i∈#M. i))",
6          "(K, (∀p∈set_of K. prime p) ∧ k = (∏i∈#K. i))"] THEN
7 OBTAIN "n = (∏i∈#(M+K). i)"
```

We will just explain the main points.

1. In line 2, we give the hint that `n` is the product over the empty multiset. After proving the hint, the program matches the resulting theorem with the pattern `∏i∈#?M. i`, obtaining a match with `?M := {#}`. This gives the goal `∀p ∈ set_of {#}. prime p`, which is easy to prove.

2. Similarly, in line 3, after proving the hint the program will try to prove `∀p ∈ set_of {#n#}. prime p`, which is easy with the assumption `prime n`.

3. Line 4, showing the existence of `m` and `k` satisfying the conjunction, is resolved using the theorem proved in Example 2, since `¬ prime n` is available and `n > 1` can be easily proved.

4. Lines 5 and 6 follow directly from the hypothesis for strong induction. On line 7, the program first proves `n = (∏i∈#(M+K). i)`, then try to prove

∀p ∈ set_of (M + K). prime p. Both parts are easy, and this finishes the theorem.

For the uniqueness statement, an intermediate lemma is needed, showing that if a prime number divides the product of a multiset of natural numbers, then it divides at least one element in that multiset. This follows by strong induction on multisets. The statement of the lemma is:

$$\texttt{prime p} \Longrightarrow \texttt{p dvd (}\prod \texttt{i}\in\texttt{\#M. i)} \Longrightarrow \exists\texttt{n. n}\in\texttt{\#M} \wedge \texttt{p dvd n}.$$

The proof script is:

```
CASE "M = {#}" THEN
CHOOSE "(M', m, M = M' + {#m#})" THEN
STRONG_INDUCT ("M", [])
```

The second line uses the result that if M is non-empty, then we can choose an element m in M, and let M' be the result of removing m from M once. From a known result, we get p divides either m or the product of M'. The second case is resolved after applying the strong induction hypothesis.

Finally, the uniqueness of factorization. The main lemma is:

$$\begin{aligned}\forall\texttt{p}\in\texttt{set\_of M. prime p} &\Longrightarrow \forall\texttt{p}\in\texttt{set\_of N. prime p}\\ &\Longrightarrow (\prod\texttt{i}\in\texttt{\#M. i) dvd (}\prod\texttt{i}\in\texttt{\#N. i)} \Longrightarrow \texttt{M} \subseteq\texttt{\# N"}\end{aligned}$$

The proof script is:

```
CASE "M = {#}" THEN
CHOOSE "(M', m, M = M' + {#m#})" THEN
OBTAIN "m dvd (∏i∈#N. i)" THEN
CHOOSES ["(n, n∈#N ∧ m dvd n)",
         "(N', N = N' + {#n#})"] THEN
OBTAIN "m = n" THEN
OBTAIN "(∏i∈#M'. i) dvd (∏i∈#N'. i)" THEN
STRONG_INDUCT ("M", [Arbitrary "N"])
```

Not much comment is needed as we have already introduced each of the ingredients. Suffice to say AUTO2 is able to fill in the gap between each of the steps. Note also that the command invoking strong induction can be placed at the end, since it merely adds the inductive hypothesis.

Finally, the main theorem:

$$\begin{aligned}\forall\texttt{p}\in\texttt{set\_of M. prime p} &\Longrightarrow \forall\texttt{p}\in\texttt{set\_of N. prime p}\\ &\Longrightarrow (\prod\texttt{i}\in\texttt{\#M. i) = (}\prod\texttt{i}\in\texttt{\#N. i)} \Longrightarrow \texttt{M = N}\end{aligned}$$

with the proof script

```
OBTAIN "M ⊆# N"
```

which tells the program to first show that M is a sub-multiset of N. After doing so, the program will understand that it needs to show N is a sub-multiset of M to finish the proof.

3 Implementation

In this section, we provide some notes on the implementation details of AUTO2, concluding with some examples of writing proof steps. Some parts of this section assumes familarity with ML and/or the core Isabelle library.

3.1 Subterms

Given a term T, the *head function* of T is either T if it is atomic, or f if T equals f applied to a list of arguments. We do not consider isolated terms of function type. So in a valid term, any f must be supplied with a full list of arguments. The *immediate subterms* of T can be thought of as the list of arguments to f, although there are a number of exceptions. These are illustrated in the examples below:

1. The immediate subterms of `if A then B else C` is [`A`] (Do not go inside branches of `if` statements).
2. The immediate subterms of `∃x. (x + a) * b` is [`a`, `b`] (Skip all terms containing bound variables).
3. The immediate subterms of `A * B * C` is [`A`, `B`, `C`] (Discard parentheses implied by the natural order of associativity. Note multiplication is declared to be left associative).
4. The immediate subterms of `A * (B * C)` is [`A`, `B * C`] (Do not discard parentheses that go against the natural order of associativity).

The restrictions in Examples 1 and 2 are intended so that branches of `if` statements and terms containing bound variables are not subjected to rewriting.

Given a term T, the *subterms* of T are its immediate subterms and subterms of its immediate subterms. The definition of immediate subterms is specified in structure `Subterms`. The structure contains a function that destructs a term T into its "skeleton" and a list of immediate subterms. For example, `A * B * C` is destructed into `?SUB * ?SUB1 * ?SUB2`, with `?SUB := A`, `?SUB1 := B`, and `?SUB2 := C`.

3.2 Rewrite table

The rewrite table is a data structure that maintains a list of equalities, and provides two functions: simplifying an expression according to the known equalities, and matching an expression against a pattern up to equivalence. The rewrite table is implemented in structure `RewriteTable`.

The core data structures in the rewrite table are the `equiv` graph and the `simp` table (`equiv` is also implemented using a table, but it is better to think of it as a graph). Each known term is represented by a node in the `equiv` graph, and an entry in the `simp` table. An edge in the `equiv` graph between (T_1, T_2), indexed by box i, means T_1 and T_2 are equal under box i. Each entry in the `simp` table corresponding to a term T is a list of pairs (i_n, T_n), where T_n is the simplest form for T, using the equalities available in box i_n.

We now describe the main functions provided by the table. For simplicity, we assume there is just one box. When there are multiple boxes, box information need to be tracked in all operations below.

3.2.1 Simplification

The *simplification* of an expression is the smallest form of the expression (under the ordering `Term_Ord.term_ord`) according to the known equalities. The *subterm simplification* of an expression is the result of simplifying each of its immediate subterms. Two expressions are *subterm equivalent* if their subterm simplifications are the same. A *head representative* of a term T, where T is not necessarily in the table, is a term T' in the table that is subterm equivalent to T. A term T in the table can be simplified by just looking up the `simp` table. A term T not in the table can be simplified as follows: first recursively simplify the immediate subterms of T, then look for a head representative T' by finding a term in the table with the same subterm simplification. If none is found, return the subterm simplification of T. Otherwise, return the simplification of T' from the `simp` table.

3.2.2 Adding new terms and equalities

When adding new terms and equalities to the rewrite table, the main issue is maintaining the following two consistency conditions:

1. If two terms are joined by an `equiv` edge, they must have the same simplification. The simplification of any term must be equal or smaller than the subterm simplification.

2. If two terms have the same subterm simplification, they must be reachable from each other in the `equiv` graph.

After adding nodes or edges to the `equiv` table that may break these conditions, we restore the conditions by updating the `simp` table (for the first condition) and adding new `equiv` edges (for the second condition). The function `process_update_simp` is responsible for maintaining the first condition, and `complete_table` for maintaining the second condition.

To add a new term T to the rewrite table, first recursively add its immediate subterms. This gives us the subterm simplification of T. Then `complete_table` is called to add `equiv` edges from T so that T is reachable in the `equiv` graph to all terms in the table with the same subterm simplification. To add a new equality, first make sure all terms occurring on the two sides of the equality are added to the table. Then adding the equality means adding a new edge to the `equiv` graph, then calling the functions maintaining the consistency conditions.

3.2.3 Matching

Given a term T, we say T' is *head equivalent* to T if it is equivalent but not subterm equivalent to T. Using the rewrite table, we can find a list of terms head equivalent to T, and are distinct up to subterm equivalence. We call this the *head equiv list* of T. The head equiv list is indexed for all terms in the table.

Given a pattern P, we say T matches P with a given assignment of schematic variables, if T is equivalent to the instantiated version of P. Given a pattern P that is not a single schematic variable, we say T *head matches* P with a given assignment of schematic variables, if T is subterm equivalent to the instantiation of P. We say two matchings are equivalent, if the assignment of each schematic variable is equivalent in the two matchings. The goal is then to find the list of matchings up to equivalence (in the case with multiple boxes, we say a matching under box i dominates a matching under box j if j is an eq-descendent of i, and if the assignment of each schematic variable is equivalent under j in the two matchings. The goal is to find the maximal matchings under this partial order).

The match and head-match functions are defined by mutual recursion. If P is an uninstantiated schematic variable, there is a unique match of T against P, instantiating that variable to T. If P is an instantiated schematic variable, we match against the instantiation instead. Otherwise, let $[T_1, ..., T_n]$ be the head equiv list of T, then the matches of T against P is the union of the head matches of each T_i against P.

The head match of T against P is computed as follows. If T is atomic, there are either zero or one matches, depending on whether T is exactly equal to P (up to type matching). Otherwise T equals a function $f(t_1, ..., t_n)$. If P is of the form $f(p_1, ..., p_n)$, then the head matches of T against P is the result of matching the ordered list of terms $[t_1, ..., t_n]$ against $[p_1, ..., p_n]$ (calling match recursively on each pair (p_i, t_i) in sequence). Otherwise there are no matches.

There are more complications introduced by abstractions (lambda terms), associative-commutative functions (match two multisets instead of two ordered lists), and the fact that sometimes we want a specific type of terms substituting a schematic variable (for example, numerical constants). See structure `RewriteTable` for the full details.

3.2.4 Incremental matching

Incremental matching (finding matches depending on a new equality) is performed with the following trick. Add a temporary primitive box i under $\{\}$ and add the new equality under box $\{i\}$. After finding all matches, filter for the matches that depend on boxes that are eq-descendents of $\{i\}$. One can then replace i with the box actually containing the new equality, to get the proper box dependence of the new matchings. This procedure is implemented in functions `append_rewrite_thm` and `replace_id_for_type`.

3.2.5 Matching an equality pattern

When one of the patterns matched by a proof step is an equality, matching against equality propositions will introduce redundancies. For example, if both $a = b$ and $a = c$ are known, then any match of pattern P against $a = b$ is also a match against $a = c$ (since $a = c$ can be rewritten to $a = b$). To fix this, we match equality patterns with terms. We say an item `TERM` T matches the equality pattern $A = B$, if T head-matches A and matches B. This logic is implemented in `fo_rewrite_match_list`.

3.2.6 Rewrite vs. simp rules (experimental)

By default, all equalities added to the rewrite table are two-way equalities: if $A = B$ (non-subterm-equivalent) is in the table, then any time A needs to be matched against a pattern, B will be considered. One can consider adding one-way equalities. For example, in the equality $n^1 = n$, one wants to consider n when matching n^1 against a pattern, but may not want n^1 to appear whenever matching n. We call the one-way equalities *simp rules* (notation `A simp= B`). Proof steps generating simp rules can be written using the function `add_simp_rule`.

We also allow registering terms c for which all equalities involving c are one-way equalities toward c. Usually c is a constant to which many terms involving variables may equal (such as 0, 1, `[]`, etc). The function `add_rew_const` adds a term to be considered a constant.

3.3 More on box items

Each non-empty box contains a list of objects of type `box_item`. The type `box_item` is defined as a record {`id: box_id, sc: int, ritem: raw_item`}. Here `id` is the ID of the box containing the item, `sc` is the score of the item, and `raw_item` contains the actual data. The datatype `raw_item` is defined as

```
datatype raw_item = Init of term * init_type
                  | FreeVar of term
                  | Handler of term * retro_handler
                  | Fact of thm
                  | MutState of state_val
```

The five possible types of raw items are:

- `Init` (t, b) specifies an initial item with term t which is considered as an assumption t if b is `InitAssum`, a goal $\neg t$ if b is `InitConcl`, and a variable t if b is `InitVar`. Both initial assumptions and conclusions must have type `Prop`. The distinction between assumptions and conclusions is used only when assembling results when a box is resolved, and for finding a good form for the induction hypothesis.

- `Freevar` v declares v to be a variable (either initial or derived from an existence fact).

- `Handler` (t, *handler*) declares that the retro-handler *handler* is responsible for removing any dependence on t in the theorem that results when a box is resolved.
- `Fact` *th* represents one of the three types of items discussed in Section 2.2. The statement (`Thm.prop_of`) of *th* is `Trueprop` P for a proposition P, `TERM` t for a term t, and $A \Longrightarrow B$ for an implication. Each hypothesis (`Thm.hyps_of`) of *th* must be either an initial assumption or a term with a registered handler, either in the current box or in one of the ancestor boxes.
- `MutState` *sval* is a mutable state. The type `state_val` is currently just `thm`, with same rules on hypothesis as for facts.

Basic manipulations of `raw_item` and `box_item` objects are defined in structure `BoxItem`.

3.4 Boxes, updates, and status

A box contains a list of box items, and its callback function (if any). Functions managing this information is defined in structure `Box`.

An object of type `raw_update` describes an update to the state. It is of four types (adding items, adding a mutable state, adding a primitive box, resolving a box). The type `update` is defined as the record

```
type update = {sc: int, prfstep_name: string,
               source: box_item list, raw_updt: raw_update}
```

Here

- `sc` is the score of the update (to be used in the priority queue).
- `prfstep_name` is the name of the proof step that produced this update.
- `source` is the list of items matched by the proof step.
- `raw_updt` is the update itself.

Functions managing `raw_update` and `update` objects are defined in structure `Update`.

The type `status` describes the state. It is defined as:

```
type status = {
  lat: BoxID.box_lattice,
  boxes: Box.box Boxidtab.table,
  shadowed: (box_id list) Termtab.table,
  mut_states: (box_item list) Symtab.table,
  queue: Updates_Heap.T,
  rewrites: RewriteTable.rewrite_table,
  ctxt: Proof.context
}
```

Here

- `lat` is a data structure recording the inheritance relations in the box lattice, as well as keeping a list of resolved boxes (implemented in structure `BoxID`).
- `boxes` is a table mapping box IDs to boxes.
- `shadowed` records information about shadowing of items due to repetition.
- `mut_states` is the table of mutable states, classified by type (represented as a string).
- `queue` is the priority queue of future updates.
- `rewrites` is the rewrite table.
- `ctxt` is the Isabelle proof context. Among other things, it maintains the list of declared variable names.

Basic manipulations of the state is defined in structure `Status`. The implementation of the main algorithm is contained in structure `ProofStatus`.

3.5 Scoring

The scoring function, computing the score of a new update in terms of the update and scores of dependent items, is contained in structure `Scores`. As it affects the order in which updates (and therefore directions of proof) are considered, it can affect a great deal the performance of the algorithm.

Currently the scoring function is very simple. In most cases, the score of the new item is the maximum score of the dependent items, plus the size (`Term.size_of_term`) of the proposition. The idea is that we are less willing to add long-winded propositions than short ones. The adjustments upon this are as follows. New variables cost 10. New boxes cost 20 plus 10 times the size of its assumptions and conclusions. Resolving a box cost -1 (so it is always done first). Implications cost 0 (since the cost is already counted when adding the $\forall$ or $\exists$ proposition producing the implication). If adding to a box that is the intersection of $n > 1$ primitive boxes, add $20(n-1)$ to the cost.

One can also let the score depend on the proof step used. Currently we implemented the simplest case: setting the cost of invoking certain proof steps to a constant number, overriding cost computations based on the content of the update (but not those based on box IDs). In general this dependence on the combination of proof steps and content can be arbitrarily complicated, reflecting the heuristic that some steps are more attractive than others. One can imagine using various machine learning techniques to automatically adjust the parameters computing the score in order to obtain the best performance.

Finally, we allow proof scripts to affect the computation of scores. Any propositions specified by proof script is given score 0 (since it is entered by a human, we strongly encourage its use). When the script shifts the focus to a box i other than $\{0\}$, the computation of scores when adding to a box j is modified as follows: instead of computing the number of primitive boxes containing j, we compute the number of primitive boxes containing j but *not* containing i. In this way, progress in boxes other than i continue (since the resolution of i may depend on case-checking in other boxes), but slowed down compared to progress directly in i.

3.6 Implementation of proof steps

In this section we discuss proof steps and the existing facility for writing them. An object of type `proofstep_fn` is the actual function the performs matching with items and producing the new `raw_update` objects. The type is defined as

```
type proofstep_fn = box_item list -> RewriteTable.rewrite_type ->
                    status -> raw_update list
```

An object of type `rewrite_type` contains a rewrite table, as well as whether the matching to be performed is an incremental matching. Only a few proof steps need to access anything in the status other than the Isabelle context `ctxt`. The exceptions include the induction proof steps, which need to access the list of initial assumptions and variables at a given box.

The type `pre_filter` is defined as

```
type pre_filter = RewriteTable.rewrite_type -> box_item -> bool
```

It represents whether it is possible for an item to serve as the first or second input to a proof step. If an item A cannot possibly be the first input to a proof step, there is no point invoking the proof step on (A, B), for any other B in the state. This is used in function `process_fact_prfsteps` for performance optimizations (`pre_filter` on proof steps taking one item as input is not very useful, since it usually takes the same amount of time for the proof step to reject the item as to run the filter).

The type `proofstep` is defined as a record

```
type proofstep =
  {name: string, filt: pre_filter list, func: proofstep_fn}
```

Here `name` is a string that uniquely identifies the proof step. `filt` is the list of pre-filters on the inputs. It must have the same length as the number of inputs the proof step takes. `func` is the actual proof step function.

In addition to defining the types, the structure `ProofStep` also provides some functions to help writing proof steps. The structure `ProofStep_Data` provides another layer of functions, directly adding proof steps to the theory using an even simpler notation. They are specially designed to add proof steps that apply a single theorem.

3.6.1 Proof steps applying a single theorem

Proof steps that directly apply a theorem can be written very simply, using functions in `ProofStep_Data`. We briefly describe the functions in this subsection.

1. (Forward reasoning) The function `add_forward_prfstep` adds a forward reasoning step from any theorem. If the theorem has one or two premises, the proof step matches that number of items with the premises and outputs the conclusion of the theorem. If the theorem has more than two premises, the proof step matches the first two premises, and output the remaining part in implication form. One restriction is that all schematic variables appearing on the output side of the proof step must appear on the input side (so all of them will have concrete values). For this reason theorems with no premises (only a conclusion) cannot be used. For this kind of theorems, one may want to use `add_known_fact`, which matches *terms* against the conclusion of the theorem, and adds the conclusion if there is a match.

2. (Backward reasoning) Theorems with one or two premises can be used directly for backward reasoning. The function `add_backward_prfstep` adds a theorem with one premise for backward reasoning, matching the conclusion and outputing the premise. The function `add_backward1_prfstep` (resp. `add_backward2_prfstep`) adds a theorem with two premises for backward reasoning, matching the conclusion and the second (resp. first) premise, and outputing the first (resp. second) premise. As with forward reasoning, any schematic variable appearing on the output side must also appear on the input side.

3. (Resolve a box) The function `add_resolve_prfstep` adds a theorem with zero or one premise for resolving a box. The proof step matches the conclusion and the premises (if there are any), and resolves the box if the theorem implies a contradiction.

4. (Rewriting rules) The function `add_rewrite_rule` adds an equality theorem (the HOL equality "=", not Pure equality "≡") for rewriting. The rewriting is in the forward direction. The function `add_rewrite_rule_back` is used for rewriting in the backward direction. The function `add_simp_rule` adds a *simplification* rule in the forward direction (see Section 3.2.6). Finally, the function `add_rewrite_rule_bidir` adds rewriting rules in both directions.

5. (Induction rules) The function `add_prfstep_induction` and its variants for strong/prop/var induction add induction rules. Generally, it adds a proof step that will automatically introduce induction under some restrictive conditions, and registers the induction rule so that scripts (see Section 2.10) can invoke it in more general situations.

All functions above (except the inductions) have a conditional form (obtained by adding suffix `_cond`). This form of the function allows adding conditions under which the proof step should be applied. The possible conditions are:

1. (Existence of a term) The function `with_term` accepts a string, parses the string to a term in the context of the theorem (with all its schematic

variables available), and returns the condition that the given term must exist in the current status. The number of `with_term` conditions plus the number of assumptions matched must be at most two (this is the “two-item matching” principle).

2. (Non-equivalence of two variables) The expression `with_cond` “`?a` $\neq$ `?b`” means the initialization of schematic variables `?a` and `?b` must be non-equivalent according to the known rewrite rules. The variables `a` and `b` must appear in the theorem.

3. (Non-equivalence between a variable and a term) The expression `with_cond` “`?a` $\neq t$” means the initialization of the schematic variable `?a` must be non-equivalent to the term t according to the known rewrite rules. The variable `a` must appear in the theorem. This and the previous condition are known as *filters*. Since they are applied after the matching, so there is no limit on the number of filters. The *plural* of `with_cond` is `with_conds`. It accepts a list of strings and returns a list of conditions.

4. (Other filters) One can write more general filters. They are converted to conditions using the function `with_filt` (with plural `with_filts`). The structure `ProofStep` contains some common filters.

3.6.2 More general proof steps

We now look at how to write more general proof steps using functions in `ProofStep`. Some of these can be written more simply using functions in `ProofStep_Data`, but we still give them as examples to illustrate the general syntax.

1. (Forward reasoning) The proof step `dvd_transitive` can be written as follows:

```
val _ = Theory.setup (
  add_gen_prfstep (
    "dvd_transitive",
    [WithFact @{term_pat "(?m::nat) dvd ?n"},
     WithFact @{term_pat "(?n::nat) dvd ?p"},
     GetFact (@{term_pat "(?m::nat) dvd ?p"},
              @{thm dvd_transitive}),
     Filter (neq_filter "m" "n"),
     Filter (neq_filter "n" "p"),
     Filter (neq_filter "m" "p")]))
```

   The function `add_gen_prfstep` accepts a pair (*name*, *descs*), where *name* is the name of the proof step, and *descs* is a list of *descriptors* specifying the proof step. It directly calls the function `gen_prfstep` to generate the proof step, and adds it to the Isabelle theory. Here the name of the proof step is `dvd_transitive`. The first two descriptors specify the patterns to be matched: `(?m::nat) dvd ?n` and `(?n::nat) dvd ?p`.

The third descriptor specify the update produced: adding a new proposition `(?m::nat) dvd ?p`. This is justified by the theorem with name `dvd_transitive`. The statement of this theorem is:

`(?m::nat) dvd ?n` $\Longrightarrow$ `?n dvd ?p` $\Longrightarrow$ `?m dvd ?p`

The function `gen_prfstep` requires that the patterns serving as inputs and output to the proof step exactly match the assumptions and conclusion of the theorem provided (up to permutation as we will see later), including names of schematic variables.

The last three lines specify the condition under which this proof step should be applied. Here `neq_filter` s_1 s_2 declares that the terms substituted for the schematic variables with names s_1 and s_2 must not be equivalent (as far as we know, according to the rewrite table). Here, if the terms substituted for `?m` and `?n` are equivalent, then the resulting update will be redundant with the second item, so we will not waste time at the main loop by producing it. The second filter is similar. The third filter is there for a different reason. If the terms substituted for `?m` and `?p` are equivalent, then the resulting proposition will be trivial. Rather than producing it, we note that in this case there is a stronger conclusion (namely `?m = ?n`), which can be produced by a different proof step.

It is simpler to add this step using functions in `ProofStep_Data`. The code is:

```
val _ = Theory.setup (
  add_forward_prfstep_cond @{thm dvd_transitive}
    (with_conds ["?m ≠ ?n", "?n ≠ ?p", "?m ≠ ?p"]))
```

2. (Backward reasoning) The proof step `coprime_dvd_mult_nat` is written as follows:

```
val _ = Theory.setup (
  add_gen_prfstep (
    "coprime_dvd_mult_nat",
    [WithFact @{term_pat "(?k::nat) dvd ?m * ?n"},
     WithGoal @{term_pat "(?k::nat) dvd ?m"},
     GetGoal (@{term_pat "coprime (?k::nat) ?n"},
              @{thm coprime_dvd_mult_nat})]))
```

This example shows the support for writing backward reasoning proof steps. Here `WithGoal` P is merely syntactical sugar for `WithFact` $\neg$ P, and likewise for `GetGoal`. The theorem `coprime_dvd_mult_nat` is:

`coprime ?k ?n` $\Longrightarrow$ `?k dvd ?m * ?n` $\Longrightarrow$ `?k dvd ?m`

By rewriting the theorem to contradiction form, reordering the assumptions, and rewriting from the contradiction form, we obtain an equivalent theorem:

`?k dvd ?m * ?n` $\Longrightarrow$ $\neg$ `?k dvd ?m` $\Longrightarrow$ $\neg$ `coprime ?k ?n`

which matches the inputs and output patterns of the proof step above. The function `gen_prfstep` automatically performs this permutation.

To add this proof step using functions in `ProofStep_Data`, we write:

```
val _ = Theory.setup (
  add_backward1_prfstep @{thm coprime_dvd_mult_nat})
```

3. (Creating a primitive box) The proof step `or_elim` is written as follows:

```
val _ = Theory.setup (
  add_gen_prfstep (
    ("or_elim",
     [WithFact @{term_pat "?A ∨ ?B"},
      CreateCase ([@{term_pat "?A::bool"}], []),
      Filter (canonical_split_filter @{const_name disj}
                  "A" "B")])))
```

   The first descriptor says the proof step should match pattern `?A ∨ ?B`. The second descriptor says when there is a match, a new primitive box should be created with initial assumption being the term substituted for `?A`. Here `CreateCase` [*assums*] [*concls*] mean creating a primitive box with list of assumptions *assums* and list of goals *concls*. The meaning of the filter is as follows: if A, B are the terms substituted for `?A`, `?B`, then A must not be a disjunction, and suppose $B = B_1 \vee \cdots \vee B_n$, then A must be smaller than each of B_i (according to `Term_Ord.term_ord`). This ensures that any disjunctive fact $C_1 \vee \cdots \vee C_n$ produces exactly one update, taking into account the associativity and commutativity of disjunction. The overall effect is that case analysis always procedes in the order given by term ordering. Note no theorem is needed to justify creating a primitive box.

   (This proof step is now replaced by mutable states. So it's here for illustration only.)

4. (Resolving a box) The proof step `less_equal_contradiction` is written as follows:

```
val _ = Theory.setup (
  add_gen_prfstep (
    "less_equal_contradiction",
    [WithFact @{term_pat "(?n::nat) < ?n"},
     GetResolve @{thm Nat.less_not_refl}]))
```

   where the theorem `Nat.less_not_refl` is `¬((n::nat) < n)`. Here the requirement is that the theorem, when written in contradiction form, has assumptions that match the input patterns up to permutation.

   Using functions from `ProofStep_Data`, we can also write:

```
val _ = Theory.setup (add_resolve_prfstep @{thm Nat.less_not_refl})
```

5. (Simplification rule) The proof step `Power.monoid_mult_class.power_one_right` is written as follows:

```
val _ = Theory.setup (
  add_prfstep_rewrite (
    [WithTerm @{term_pat "(?a::nat) ^ 1"},
     Filter (neqt_filter "a" @{term "1::nat"})],
    @{thm power_one_right}))
```

The function `add_prfstep_rewrite` makes it easy to write proof steps that applies a rewrite rule. The name of the proof step is taken from the name of the theorem. The statement of the theorem is

```
  ?a ^ 1 = ?a
```

We require the term pattern to agree with one side of the rewrite rule, up to type matching (the proof step is still only applied when `?a` is a natural number). The proof step then produces an equality from the term pattern to the other side. The second descriptor gives a filter: the term substituted for `?a` should not be equivalent to `1::nat`, since this case is covered by another proof step.

Using functions in `ProofStep_Data`, we can also write:

```
val _ = Theory.setup (
  add_simp_rule_cond @{thm power_one_right} [with_cond "?a ≠ 1"])
```

6. (Conversions) The proof step `rewrite_not_forall` is written as follows:

```
val _ = Theory.setup (
  add_prfstep_conv (
    "rewrite_not_forall",
    [WithTerm @{term_pat "¬(∀x. ?A)"}],
    (Conv.rewr_conv (to_meta_eq @{thm HOL.not_all}))))
```

Sometimes applying a rewrite rule requires second order matching. In this case `add_rewrite_rule` will not work, since the specified pattern is first order. A more general function `add_prfstep_conv` should be used. This function adds a proof step that matches the given pattern and applies the conversion. Since a `conv` object is opaque, it is not possible to check whether it can be applied beforehand. It is considered an error if the conversion fails, or produces a trivial equality. In this way, we require the condition under which a proof step will be applied be specified clearly in the text of the proof step.

The functions creating the proof steps for the setup functions `add_prfstep_conv` and `add_rewrite_rule` are `prfstep_conv` and `prfstep_rrule` respectively.

7. (Adding an implication) The proof step `exists_intro2` is written as follows:

```
val _ = Theory.setup (
  add_prfstep_thm (
    "exists_intro2",
    [WithGoal @{term_pat "∃x y. ?P ∧ ?Q"},
     Filter (ac_atomic_filter @{const_name conj} "P"),
     Filter (subset_var_filter "Q" "P")],
    @{thm exists_intro2}))
```

The theorem `exists_intro2` is:

`¬(∃x y. P x y ∧ Q x y) ⟹ P x y ⟹ ¬(Q x y).`

The function `add_prfstep_thm` calls `prfstep_thm` to produce a proof step that matches items against the pattern `¬(∃x y. P x y ∧ Q x y)`, and for every match that satisfies the two given filters, applies the theorem `exists_intro2` to produce an implication `P x y ⟹ ¬(Q x y)`. Hence it is similar to `add_gen_prfstep`, except it allows second order matching (using Isabelle's `MRS` function). The proof step can be read as a backward reasoning step as follows: if the goal is to show the existence of x, y satisfying predicates P and Q, and there are terms x, y satisfying P, then it suffices to show that they also satisfy Q.

The first filter, generated by function `ac_atomic_filter`, declares that P itself must not be a conjunction. The second filter says that any bound variable mentioned by Q must also be mentioned by P. In this case, it means P must be a predicate on both x and y. (In the instantiation of P and Q returned by the matching function, bound variables are replaced by schematic variables. Hence the function `subset_var_filter` actually checks the set of schematic variables).

As an example, we reproduce an initial assumption (negated initial goal) in `not_prime_eq_prod_nat`:

`¬ ∃m k. n = m * k ∧ 1 < m ∧ m < n ∧ 1 < k ∧ k < n.`

The body of the existence statement is a conjunction of five terms. Without the two filters, `exists_intro2` will produce 30 implications (P can be any non-empty proper subset of the five terms). With the first filter, the possibilities are reduced to five, with P being each of the five terms in the conjunction. However, only one of these five terms mention both `m` and `k`, so after the second filter, the proof step generates only one implication, namely

`n = ?x * ?y ⟹ ¬(1 < ?x ∧ ?x < n ∧ 1 < ?y ∧ ?y < n).`

which is the one needed for the proof.

8. (Using a trivial fact) The proof step `exists_n_dvd_n` is written as follows:

```
val _ = Theory.setup (
  add_prfstep_two_stage (
    "exists_n_dvd_n",
    [WithGoal @{term_pat "∃k. k dvd (?n::nat) ∧ ?A"},
     GetFact (@{term_pat "(?n::nat) dvd ?n"}, @{thm n_dvd_n})],
    @{thm exists_intro}))
```

The meaning of the proof step is as follows: if we want to show the existence of k such that k divides n, and satisfies another predicate A, then it suffices to show that n satisfies A. This uses the trivial fact `n dvd n`. Usually, it is not good to add trivial facts like this to the main state, since they are often the source of meaningless derivations. Instead, we want to generate the trivial fact, and use it only once, in this case on the existence goal. The proof step writing function `add_prfstep_two_stage` makes it possible. The first descriptor specifies the pattern to be matched. The second descriptor specifies that a trivial fact `n dvd n` should be generated, relying on the theorem `n_dvd_n` (in this case, the statement of the theorem should exactly agree with the pattern producing the trivial fact). Next, the theorem `exists_intro` is applied to the matched item and the trivial fact (in that order). The theorem `exists_intro` is

`¬(∃x. P x ∧ Q x) ⟹ P x ⟹ ¬(Q x)`

Hence the existence goal will be matched against `¬(∃x. P x ∧ Q x)`, and the trivial fact will be matched against `P x`, with `¬(Q x)` produced as a result. As an example, suppose we have goal `∃p. p dvd n ∧ prime p` in box $\{0\}$. Using trivial fact `n dvd n`, we see the proof will be finished if `prime n` holds. Hence we can add `¬ prime n` to box $\{0\}$ (this is one step in the proof of the theorem `¬(n = 1) ⟹ ∃p. p dvd n ∧ prime p`).

9. (Arithmetic proof steps) In principle, any Isabelle tactic can be packaged as a proof step, by letting the proof step invoke the tactic on any facts to which the tactic is intended to be applied. In practice, one should avoid packaging any tactic (or functionality of a tactic) that involves searching, which is best left to the main loop. The tactics that are ideal for packaging are those that perform definite, algorithmic computations.

   In this example, we show how to write proof steps that perform arithmetic computation by packaging the Isabelle tactic `arith_tac`. First, some definitions to shorten the code:

```
fun lookup_numc inst n = dest_numc (lookup_instn inst ("NUMC", n))
fun mk_nat n = HOLogic.mk_number HOLogic.natT n
fun prove_by_arith ctxt ths goal =
    let
      val goal' = Logic.list_implies (
            map Thm.prop_of ths, HOLogic.mk_Trueprop goal)
    in
      ths MRS (Goal.prove ctxt [] [] goal' (K (Arith_Data.arith_tac ctxt 1)))
    end
fun contra_by_arith ctxt ths = prove_by_arith ctxt ths @{term "False"}
```

   The proof step `compare_consts` is written as follows:

```
val _ = Theory.setup (
  add_prfstep_thm_fn (
    "compare_consts",
    [WithFact @{term_pat "(?NUMC1::nat) = ?NUMC2"},
     Filter (fn _ => fn (_, inst) =>
                 lookup_numc inst 1 <> lookup_numc inst 2)],
    fn ctxt => fn (_, ths) => contra_by_arith ctxt ths))
```

   This proof step takes a proposition $m = n$, where both m and n are numerical constants (enforced by the special name `NUMC` of the schematic variables), and derive a contradiction if m and n are different constants.

The filter declares that the ML-value of m and n must be unequal, and the body of the proof step uses `arith_tac` to produce a theorem deriving the contradiction (the theorem returned by the body is `False`, which `add_prfstep_thm_fn` will use the produce an update resolving the box.

This example shows that a proof step can be arbitrarily complicated, deconstructing the matched item into ML-values, and performing computations using them. The only requirement is that the final results must be verified by a theorem.

For a more complex example, let us look at a proof step simplifying a comparision involving constants.

```
val _ = Theory.setup (
  add_prfstep_thm_fn (
    "le_plus_consts_fact'",
    [WithFact @{term_pat "(?NUMC1::nat) + ?a <= ?NUMC2"},
     Filter (fn _ => fn (_, inst) => lookup_numc inst 1 <> 0)],
    (fn ctxt => fn ((_, inst), ths) =>
      let
        val (n1, n2) = (lookup_numc inst 1, lookup_numc inst 2)
      in
        if n1 <= n2 then
          let
            val inst' = update_env (("DIFF", 0), mk_nat (n2 - n1)) inst
            val res_pat = @{term_pat "(?a::nat) <= ?DIFF"}
          in
            prove_by_arith ctxt ths (Envir.subst_term inst' res_pat)
          end
        else contra_by_arith ctxt ths
      end)))
```

This proof step matches facts of the form $n_1 + a \leq n_2$ where n_1 and n_2 are numerical constants, and a is an arbitrary term. If $n_1 > n_2$, then this implies a contradiction. If $0 < n_1 \leq n_2$, then this fact can be simplified to $a \leq n_2 - n_1$.

10. (Existence elimination and induction) There are still more complicated proof steps, involving retro-handlers or box callbacks. We refer directly to the code for details. Instantiation of an existence fact is implemented in function `exists_elim` as a part of structure `Logic_ProofSteps`. Simple induction and strong induction are implemented using retro-handlers, in functions `prfstep_induction` and `prfstep_strong_induction` in structure `Induct_ProofSteps`. Finally, general induction on inductively defined propositions and types are implemented using box callbacks, in functions `prfstep_prop_induction` and `prfstep_var_induction` in the same structure.