

AUTO2 Documentation (draft)

Bohua Zhan

September 19, 2015

Contents

1	Introduction	2
2	Overview of algorithm	2
2.1	Box lattice	2
2.2	Box items	3
2.3	Proof steps	4
2.4	Rewriting and matching	7
2.5	The main loop	8
2.6	Normalization (experimental)	9
2.7	Retro-handlers and induction (experimental)	9
2.7.1	Using an existence fact	9
2.7.2	Induction	10
2.8	Box callbacks	10
2.9	Proof scripts	11
2.10	Examples in number theory	12
2.10.1	Example 1: <code>prime_imp_coprime_nat</code>	12
2.10.2	Example 2: <code>not_prime_eq_prod_nat</code>	13
2.10.3	Example 3: <code>prime_power_mult</code>	14
2.10.4	Example 4: <code>prime_factor_nat</code>	16
2.10.5	Example 5: Infinitude of prime numbers	17
2.10.6	Example 6: Unique factorization theorem	18
3	Implementation	20
3.1	Subterms	20
3.2	Rewrite table	20
3.2.1	Simplification	21
3.2.2	Adding new terms and equalities	21
3.2.3	Matching	22
3.2.4	Incremental matching	22
3.2.5	Matching an equality pattern	23
3.3	More on box items	23
3.4	Boxes, updates, and status	24

3.5	Scoring	25
3.6	Implementation of proof steps	25
3.6.1	Proof steps applying a single theorem	26
3.6.2	More general proof steps	28
3.6.3	Standard library of proof steps	32

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, $[A, B] \implies C$ is first transformed into $[A, B, \neg C] \implies \text{False}$). 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 set 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 $A \implies \mathbf{False}$, 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 $AD \implies \mathbf{False}$ will resolve box $\{1\}$, and showing $BC \implies \mathbf{False}$ 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 taking the union of the two sets 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, representing what is known in that box. Each item is specified by a *type* and a list of terms called *tname*, and possibly backed by a formal theorem. New item types can be added by the user. Currently the possible types of items are:

- **VAR** v : free variable v is introduced in this box. No theorem is needed.
- **TERM** t : term t is present in this box. No theorem is needed.
- **PROP** P : proposition P can be proved from the assumptions in this box. Backed by a formal theorem with statement P .
- **EQ** $[s, t]$: terms s and t are equivalent in this box. Backed by a formal theorem with statement $s = t$.
- **DISJ** $[P_1, \dots, P_n]$: each P_i is a proposition possibly containing schematic variables (Isabelle for arbitrary variables). At least one of P_i is true in this box for any instantiation of the schematic variables. Backed by a formal theorem with statement $P_1 \vee \dots \vee P_n$.

- **DISJ_ACTIVE** $[P_1, \dots, P_n]$: same as **DISJ**, except when there are no schematic variables, the prover will attempt to derive a contradiction from $P_1 \vee \dots \vee P_n$ by case checking.
- **NAT_ORDER** $[x, y, n]$: here n is an integer, and x, y are natural numbers. Shows $x \leq y + n$ if $n \geq 0$ and $x + n \leq y$ if $n < 0$. Backed by a theorem of the same form.

Except for **VAR** and **TERM**, all remaining types represent propositions that can be proved in the given box. There are more types than just **PROP** so that we can specify different behavior for the prover on different kinds of propositions. For example, the equalities (type **EQ**) can be used for matching, and the active disjunctions (type **DISJ_ACTIVE**, but not **DISJ**) can be used for case checking, etc. Note also that since we work in classical logic, with the goal of deriving a contradiction, the presence of proposition P in a box means proving $\neg P$ resolves this box, hence $\neg P$ can be thought of as one of the goals 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 $[Q_1, \dots, Q_i]$, 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 represent the smallest units of action in the proving process. 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 creates a list of *updates* that can be applied to the state. The update is usually some kind of modification to the box that is the intersection of the boxes containing the matched items (or a descendent box due to rewriting, to be discussed later). There are four types of updates:

- Add a new item: if the item needs to be backed by a theorem, the proof step should derive that theorem, using the theorems backing the matched items. This is the most common kind of operation, representing a step of reasoning from known facts to a new fact.
- Create a new primitive box: this lets the prover to consider a particular case of the result to be proved. This should be used sparingly, only when it seems apparent that case checking is necessary. Non-apparent case checking can be specified by the user (see Section 2.9).

- Resolve a box: shows a contradiction exists in the given box. The proof step should derive a contradiction from the theorems backing the matched items.
- Shadow an item: declares that one of the matched items is extraneous in the given box. The shadowed item will not be used in any further matching in proof steps. Usually an item is shadowed because it is trivial or redundant to another item. No theorem is needed to back a shadowing. Nevertheless, this should be used carefully, only after being certain that matching the item is no longer necessary.

Proof steps can be added by the user. AUTO2 provides utilities for concisely specifying various common types of proof steps. More complicated proof steps can also be written directly in ML.

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 term of the right type. When creating a new item (or primitive box), schematic variables in the result pattern are instantiated to the corresponding terms (even if the uninstantiated pattern also makes sense).

1. (Forward reasoning) If two items match $?m \text{ dvd } ?n$ and $?n \text{ dvd } ?p$, produce a new item $?m \text{ dvd } ?p$ (divisibility is transitive).
2. (Backward reasoning) If two items match $?k \text{ dvd } ?m * ?n$ and $\neg ?k \text{ dvd } ?m$, produce a new item $\neg \text{coprime } ?k ?n$. 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 $?k \text{ dvd } ?m * ?n$ and needing to prove $?k \text{ dvd } ?m$, it suffices to show that $?k$ and $?n$ are coprime.
3. (Simplification rule) If an item matches $\text{TERM } ?a^0$, add new item $?a^0 = 1$. This proof step acts as a simplification rule.
4. (Expanding a definition) If an item matches $\text{TERM } \text{prime } ?p$, add new item $\text{prime } ?p = (1 < ?p \wedge \forall m. m \text{ dvd } ?p \longrightarrow m = 1 \vee m = ?p)$.

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 $\text{TERM } \text{gcd } ?a ?b$, add new item $\text{gcd } ?a ?b \text{ dvd } ?a$. 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 $?a \neq ?b$, where $?a$ and $?b$ are boolean variables, create a new primitive box, inheriting from the box containing $?a \neq ?b$, with assumption $?a$ and conclusion $?b$. This allows proving an if-and-only-if statement by first trying to prove the forward direction.
8. (Adding a disjunction) If an item matches $\forall x. ?A(x) \longrightarrow ?B(x)$, add disjunction $\text{DISJ } [\neg ?A(?x), ?B(?x)]$. Associative property of disjunction is automatically applied. For example, given the item

$$\forall m. m \text{ dvd } p \longrightarrow m = 1 \vee m = p,$$

add the disjunction $\text{DISJ } [\neg ?m \text{ dvd } p, ?m = 1, ?m = p]$.

9. (Resolving a disjunction) For any disjunction $\text{DISJ } [P_1, \dots, P_n]$, and a proposition A , match A with each of the patterns $\neg P_i$ that contains the largest number of schematic variables. Produce new disjunction instantiating the matched schematic variables and removing P_i (or ordinary proposition if there is just one term remaining in the disjunction and no schematic variables). For example, with disjunction $\text{DISJ } [\neg ?m \text{ dvd } p, ?m = 1, ?m = p]$, and item $n \text{ dvd } p$, produce new item $\text{DISJ } [n = 1, n = p]$.
10. (Creating a case from disjunction) Given active disjunction $\text{DISJ_ACTIVE } [P_1, \dots, P_n]$, containing no schematic variables, create a new primitive box with assumption P_1 . Note there is no corresponding proof step for DISJ items, so the prover will behave differently whether a disjunction is added as DISJ or DISJ_ACTIVE (generally, DISJ is added from implications, and DISJ_ACTIVE is added from conjunctive goals and disjunctive facts). Note also that unless the new box is resolved, removing P_1 , no cases are created for P_2, \dots, P_n .
11. (Arithmetic) Given item $?a + n_1 \leq ?b + n_2$, where $n_1 \leq n_2$ are constant natural numbers, produce new item $?a \leq ?b + (n_2 - n_1)$, where the subtraction is performed. For example, with $a + 3 \leq b + 7$, obtain $a \leq b + 4$.

Hence items that are propositions 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 of the system as output. In general, any computation following a definite algorithm, and whose results can be described concisely in usual mathematical language, should be written as a single proof step (anything involving searching, however, is probably best left outside of proof steps).

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.

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. As output, the rewrite table answers two kinds of queries: it finds the simplest known form of a given expression, and it produces all matches of an expression against a pattern, up to rewriting the expression using the known equalities. The matching is essentially first order, although it can also handle some second order patterns, where any schematic variable in functional position is applied only to distinct bound variables. Using the simplify function, we can also check whether any two expressions are known to be equivalent. This interface is essentially the same as for *E-matching* in SMT-solvers, although here we also have to deal with different cases, organized as a box lattice.

As an example, suppose $\text{prime } p = (1 < p \wedge \forall m. m \text{ dvd } p \longrightarrow m = 1 \vee m = p)$ is added to the rewrite table. Then the term $\neg \text{prime } p$ matches the pattern $\neg(?A \wedge ?B)$, with $?A := 1 < p$ and $?B := \forall m. m \text{ dvd } p \longrightarrow m = 1 \vee m = p$. This allows de Morgan’s law to be applied to $\neg \text{prime } p$. 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 to the state (except the initial one) comes from a proof step, and contains one of four types of actions: adding items to a box, creating a primitive box, resolving a box, and shadowing an item. The first two types of updates are inserted into the priority queue, with priority equal to the score of the items or boxes to be added (smaller value means higher priority). The latter two types of updates are performed immediately, since they always simplify the state.

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 or 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 either performed immediately or added to the priority queue, depending on its type.
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 either performed immediately or 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.

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 $A_1 \wedge \dots \wedge A_n$, split into items A_1, \dots, A_n .
- For item of form $\neg(A_1 \vee \dots \vee A_n)$, split into items $\neg A_1, \dots, \neg A_n$.
- Replace $A = \text{True}$ and $A \neq \text{False}$ by A . Replace $A = \text{False}$ and $A \neq \text{True}$ by $\neg A$.
- For associative-commutative operations, put in normal form (remove unnecessary parentheses and put arguments in increasing order).
- 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 simplifications are not added to the rewrite table.

2.7 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. There are two main examples of this in AUTO2: using an existence fact, and induction.

2.7.1 Using an existence fact

If $\exists x.P(x)$ 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 does 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

$$\exists x.P(x) \implies \forall x.(P(x) \longrightarrow Q) \implies Q$$

(**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 $\forall x.(P(x) \longrightarrow \text{False})$ will be available to the parent box, so 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.

2.7.2 Induction

Some forms of induction rules can be supported using retro-handlers (those that cannot are supported through box callbacks, see Section 2.8). One particularly common form is:

$$P(t_o) \implies \forall t. (P'(t) \longrightarrow P(t)) \implies \forall t. P(t), \quad (1)$$

where t_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: $P(0) \implies \forall n. (P(n-1) \longrightarrow P(n)) \implies \forall n. P(n)$.
- For lists: $P([]) \implies \forall l. (P(\text{tl } l) \longrightarrow P(l)) \implies \forall l. P(l)$.
- For trees: $P(\text{leaf}) \implies \forall t. (P(\text{lsub } t) \wedge P(\text{rsub } t) \longrightarrow P(t)) \implies \forall t. P(t)$.

After verifying the base case, we can apply an induction rule of this form by adding an assumption to the box corresponding to the previous case. 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 $[P_1(n), \dots, P_i(n)]$ 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 $P(n)$: $[P_1(n), \dots, P_i(n)] \implies \text{False}$. 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 $P(n-1) \implies P(n)$ 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 actually a subcase of the above. The strong induction rule, for natural numbers or any set that is well ordered, is given by

$$(\forall n. ((\forall m < n. P(m)) \longrightarrow P(n)) \implies \forall n. P(n).$$

This is in the form of Equation 1, without a base case and with $P'(n) = \forall m < n. P(m)$.

2.8 Box callbacks

Box callbacks allow one to specify a procedure to be performed 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 outputting 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*, which is a theorem of the form $A_1 \implies \dots \implies A_n \implies B$, where each A_i is of the form

$$\bigwedge x_{i1} \dots x_{ik}. A_{i1} \implies \dots \implies A_{im} \implies B_i.$$

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 .

Induction principles of inductively defined propositions and types often do not have a simple form allowing the use of retro-handlers. In this case box callbacks are used to apply them. See the development of Hoare logic for examples.

2.9 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 the 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** $[s_1, \dots, s_n]$ is equivalent to **CHOOSE** s_1 **THEN** \dots **THEN** **CHOOSE** s_n .
- **INDUCT** $(n, [\text{OnFact } n \neq n_o, \text{Arbitrary } m])$. Apply induction rule as described in Section 2.7. Here n_o is the value of the base case of the induction. Wait for proposition $n \neq n_o$ 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 **Arbitrariables** is used if multiple variables are held arbitrary. This and all induction commands below require the proper induction rule to be registered.

- **STRONG_INDUCT** ($n, [\dots]$), **VAR_INDUCT** ($n, [\dots]$). Perform strong induction (Section 2.7) or general variable induction (Section 2.8). Arbitrary variables can be specified.
- **PROP_INDUCT** ($exp, [\dots]$). With exp as an initial assumption, perform propositional induction on exp . The proposition exp should be in the form $f(x_1, \dots, x_n)$, 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.

2.10 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 steps 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.

2.10.1 Example 1: `prime_imp_coprime_nat`

We begin with a relatively simple result, requiring no case checking. The statement of the theorem is

$$[\text{prime } p, \neg p \text{ dvd } n] \implies \text{coprime } p \ n$$

In Isabelle, `coprime p n` is abbreviation for `gcd p n = 1`. Hence the goal is to derive a contradiction from the list of assumptions `prime p`, `$\neg p \text{ dvd } n$` , and `$\text{gcd } p \ n \neq 1$` .

1. From **TERM** `prime p`, add `prime p = (1 < p \wedge $\forall m. m \text{ dvd } p \longrightarrow m = 1 \vee m = p$)` to box 0.
2. Using equality from the previous step, `prime p` matches pattern `?A \wedge ?B`, which adds `1 < p` and `$\forall m. m \text{ dvd } p \longrightarrow m = 1 \vee m = p$` to box 0.

3. From $\forall m. m \text{ dvd } p \longrightarrow m = 1 \vee m = p$, add $\text{DISJ } [\neg ?m \text{ dvd } p, ?m = 1, ?m = p]$.
4. From $\text{TERM } \text{gcd } p \ n$, add basic properties of gcd: $\text{gcd } p \ n \text{ dvd } p$ and $\text{gcd } p \ n \text{ dvd } n$.
5. The negation of $\text{gcd } p \ n \text{ dvd } p$ matches the first term of disjunction $\text{DISJ } [\neg ?m \text{ dvd } p, ?m = 1, ?m = p]$. This produces $\text{DISJ } [\text{gcd } p \ n = 1, \text{gcd } p \ n = p]$.
6. The negation of $\text{gcd } p \ n \neq 1$ matches the first term of $\text{DISJ } [\text{gcd } p \ n = 1, \text{gcd } p \ n = p]$. This results in item $\text{gcd } p \ n = p$.
7. However, with $\text{gcd } p \ n = p$, there is a contradiction between assumption $\neg p \text{ dvd } n$ and derived fact $\text{gcd } p \ n \text{ dvd } n$. This proves the theorem.

2.10.2 Example 2: not_prime.eq_prod_nat

In this example, we show how to use case checking from DISJ_ACTIVE to prove a conjunction of goals. The statement of the theorem is

$$[n > 1, \neg \text{prime } n] \implies \exists m \ k. n = m * k \wedge 1 < m \wedge m < n \wedge 1 < k \wedge k < n.$$

The goal is to derive a contradiction from the list of assumptions $n > 1$, $\neg \text{prime } n$, and $\neg \exists m \ k. n = m * k \wedge 1 < m \wedge m < n \wedge 1 < k \wedge k < n$.

1. From $\text{TERM } \text{prime } n$, add $\text{prime } n = (1 < n \wedge \forall m. m \text{ dvd } n \longrightarrow m = 1 \vee m = n)$.
2. $\neg \text{prime } n$ matches the pattern $\neg(?A \wedge ?B)$. Applying de Morgan's law, we get item $\text{DISJ_ACTIVE } [\neg 1 < n, \neg \forall m. m \text{ dvd } n \longrightarrow m = 1 \vee m = n]$.
3. The negation of assumption $n > 1$ matches the first term of the DISJ_ACTIVE item from the previous step. This produces term $\neg \forall m. m \text{ dvd } n \longrightarrow m = 1 \vee m = n$.
4. $\text{TERM } \neg \forall m. m \text{ dvd } n \longrightarrow m = 1 \vee m = n$ matches pattern $\text{TERM } \neg \forall x. ?A(x)$, adding an equality rewriting it to $\exists m. \neg(m \text{ dvd } n \longrightarrow m = 1 \vee m = n)$.
5. The result of previous step matches pattern $\exists x. ?A(x)$. Using the existence fact, we may add a new variable m and fact $\neg(m \text{ dvd } n \longrightarrow m = 1 \vee m = n)$.
6. From the previous step, it suffices to show $m \text{ dvd } n \longrightarrow m = 1 \vee m = n$. So $m \text{ dvd } n$ holds and it suffices to show $m = 1 \vee m = n$ (that is, add $\neg(m = 1 \vee m = n)$ as a fact). Applying de Morgan's law to the latter, add $m \neq 1$ and $m \neq n$ as facts.

7. Item $m \text{ dvd } n$ from the previous step rewrites to $\exists k. n = m * k$. Using the existence statement, add new variable k and item $n = m * k$.
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. From the assumption $\neg \exists m k. n = m * k \wedge 1 < m \wedge m < n \wedge 1 < k \wedge k < n$, add `DISJ_ACTIVE` [$n \neq ?m * ?k, \neg 1 < ?m, \neg ?m < n, \neg 1 < ?k, \neg ?k < n$]. The first term of this item matches the negation of $n = m * k$, giving `DISJ_ACTIVE` [$\neg 1 < m, \neg m < n, \neg 1 < k, \neg k < n$].
10. From the previous `DISJ_ACTIVE` item, add primitive box 1 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 proposition $1 < m$ and new item `DISJ_ACTIVE` [$\neg m < n, \neg 1 < k, \neg k < n$] are added to box $\{0\}$.
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 $m < n$ and new item `DISJ_ACTIVE` [$\neg 1 < k, \neg k < n$] to box $\{0\}$.
13. Primitive box 3 is created under $\{0\}$ with assumption $\neg 1 < k$. Since $1 \leq k$ is known from Step 8, we get $1 = k$ in box $\{3\}$.
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 $1 < k$ and $\neg k < n$ as facts to $\{0\}$.
15. Since $k \leq n$ is known from Step 8, we get $k = n$. But this 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.

2.10.3 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

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

Since this proof involves induction with arbitrary variables, which is not automatically tried, we need to provide the following proof script, indicating an induction on k , with x and y held arbitrary.

```
1 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 induction on k and case analysis on $k = 0$ and $k \neq 0$.
2. Add equality rewriting the goal to $\exists i. x = p^i \wedge \exists j. y = p^j$. This produces item `DISJ_ACTIVE` $[\neg \exists i. x = p^i, \neg \exists j. y = p^j]$.
3. Create primitive box 2 under box $\{0\}$ with goal $\exists i. x = p^i$. 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 item `DISJ` $[\neg ?x * ?y = p^{k-1}, \exists i j. ?x = p^i \wedge ?y = p^j]$.
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 $[\text{prime } p, p \text{ dvd } m * n] \implies p \text{ dvd } m \vee p \text{ dvd } n$, even if the second condition is hidden as an equality $p * q = m * n$). Then we have `DISJ_ACTIVE` $[p \text{ dvd } x, p \text{ dvd } y]$. Primitive box 3 is added under $\{0\}$, with assumption `p dvd x`.
7. In box $\{3\}$, we get variable ka and fact $x = ka * p$ from `p dvd x`.
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 previous step matches the negation of the first term in the `DISJ` item 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 from the previous step are used to add variables i, j , and propositions $ka = p^i$ and $y = p^j$ to $\{3\}$.
11. The term $ka * p$ matches pattern $?p * ?p^{?i}$, 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 item `DISJ` $[\neg x = p^{?i}]$ (this is added as a `DISJ` item because it contains a schematic variable). Since the result from previous step matches the pattern $x = p^{?i}$, box $\{2, 3\}$ is resolved.

13. Resolving box $\{2, 3\}$ adds facts $\exists i. x = p^i$ to box $\{3\}$ and $\neg p \text{ dvd } x$ 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 item `DISJ` $[\neg y = p^{?j}]$. Since $y = p^j$ (from Step 10) matches the left side of this implication, box $\{3\}$ is resolved. Facts $\neg p \text{ dvd } x$ and then $p \text{ dvd } y$ are added to the home box $\{0\}$.
15. Checking the case $p \text{ dvd } y$ proceeds in box $\{0\}$ in the same way as checking the case $p \text{ dvd } x$ in box $\{3\}$, again using the case analysis provided by primitive box 2. The rest is omitted.

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

2.10.4 Example 4: `prime_factor_nat`

Starting with this example, we will mostly focus on proof scripts, and omit details at the 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

$$n \neq 1 \implies \exists p. p \text{ dvd } n \wedge \text{prime } p.$$

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 $\forall m < n. m \neq 1 \implies (\exists p. \text{prime } p \wedge p \text{ dvd } m)$ to box $\{0\}$. The rest of the proof proceeds as follows:

1. Since $n \text{ dvd } n$, we obtain the fact $\neg \text{prime } n$. This gives $\neg(1 < n) \vee \neg(\forall m. m \text{ dvd } n \implies m = 1 \vee m = n)$.
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 $p \text{ dvd } 0$ holds for any p , it suffices to show $\exists p. \text{prime } p$. But this is a known result.
3. In the second case, instantiate with variable m and proposition $\neg(m \text{ dvd } n \implies m = 1 \vee m = n)$. This becomes a conjunction of facts $m \text{ dvd } n$, $m \neq 1$, and $m \neq n$. From $m \text{ dvd } n$ we get $m \leq n$, which when combined with $m \neq n$ gives $m < n$. From the strong induction hypothesis, we get $\exists p. \text{prime } p \wedge p \text{ dvd } m$. Instantiate with variable p and propositions $\text{prime } p$ and $p \text{ dvd } m$. From $\text{prime } p$ it suffices to show $p \text{ dvd } n$, but this follows from $p \text{ dvd } m$ and $m \text{ dvd } n$.

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 $\exists p. p \text{ dvd } ?n \wedge \text{prime } p$, and for each match, creates a goal $?n \neq 1$ (with instantiated $?n$). This proof step will be used in the next example.

2.10.5 Example 5: Infinitude of prime numbers

In this example, 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 with just a few lines of hints, the computer is able 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

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

We use the following proof script for the lemma:

```

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

```

The proof proceeds as follows:

1. The first command creates a new box with goal $\exists p. \text{prime } p \wedge p \text{ dvd fact } n + 1$ (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 $\text{fact } n + 1 \neq 1$. Alternatively, we have $\text{fact } n + 1 = 1$. This becomes $\text{fact } n = 0$, which contradicts the fact $\text{fact } n \geq 1$ (which is added on seeing the term **fact** n). This resolves the new box. The resulting existence statement is instantiated with variable p and propositions **prime** p and $p \text{ dvd fact } n + 1$.
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 $p \leq n$, and under that a new box with goal $p \text{ dvd fact } n$. By backward reasoning with theorem $[1 \leq p, p \leq n] \implies p \text{ dvd fact } n$, it suffices to show $1 \leq p$. But we have $p > 1$ from **prime** p . This resolves the goal $p \text{ dvd fact } n$. From $p \text{ dvd fact } n$ and $p \text{ dvd fact } n + 1$, we obtain $p \text{ dvd } 1$, which means $p = 1$, contradicting $p > 1$. This resolves the box with assumption $p \leq n$.
3. The new fact $\neg p \leq n$ 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** " $p \text{ dvd fact } n$ ", since the goal of proving $n < p$ already provides the assumption " $p \leq n$ ". 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 $\exists p. \text{prime } p \wedge ?n < p$ will be resolved.

Now we come to the main theorem. The statement is

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

The proof script has only one line:

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

This command adds a new box with goal $\neg(\exists b. \text{prime } b \wedge \text{Max } \{p. \text{prime } p\} < b)$. 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 $\text{prime } b$ and $\text{Max } \{p. \text{prime } p\} < b$. From the latter fact, we obtain $b \notin \{p. \text{prime } p\}$, which contradicts $\text{prime } b$. This proves the theorem.

2.10.6 Example 6: Unique factorization theorem

In this example, 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:

$$n > 0 \implies \exists M. (\forall p \in \text{set_of } M. \text{prime } p) \wedge n = (\prod_{i \in \#M. i}).$$

Here M ranges over finite multisets of natural numbers. The term `set_of` M is the set of elements in M (forgetting multiplicity). The notation $\prod_{i \in \#M. i}$ means “the product of i when i ranges over M , counting multiplicity”.

The proof script is below. Here $\in\#$ is the membership operator for multisets, $\{\#\}$ is the empty multiset, and $\{\#n\}$ is the multiset with a single n .

```
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 $n = \prod_{i \in \#?M. i}$, obtaining a match with $?M := \{\#\}$. This gives the goal $\forall p \in \text{set_of } \{\#\}. \text{prime } p$, which is easy to prove. This adds $n \neq 1$ and then $n > 1$ as facts.
2. Similarly, in line 3, after proving the hint the program will try to prove $\forall p \in \text{set_of } \{\#n\}. \text{prime } p$, which is easy with the assumption $\text{prime } n$. This adds $\neg \text{prime } n$ as fact.
3. Line 4, showing the existence of m and k satisfying the conjunction, is resolved using the theorem proved in Example 2, since both $\neg \text{prime } n$ and $n > 1$ are available.

4. Lines 5 and 6 follow directly from the strong induction hypothesis (the purpose of these two lines is to fix names of variables M and K for line 7). On line 7, the program first proves $n = (\prod_{i \in \#(M+K)} i)$, then try to prove $\forall p \in \text{set_of } (M + K). \text{ prime } p$. 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:

$$\text{prime } p \implies p \text{ dvd } (\prod_{i \in \#M} i) \implies \exists n. n \in \#M \wedge p \text{ dvd } n.$$

The proof script is:

```

1 CASE "M = {#}" THEN
2 CHOOSE "(M', m, M = M' + {#m#})" THEN
3 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:

$$\begin{aligned} \forall p \in \text{set_of } M. \text{ prime } p &\implies \forall p \in \text{set_of } N. \text{ prime } p \\ &\implies (\prod_{i \in \#M} i) \text{ dvd } (\prod_{i \in \#N} i) \implies M \subseteq_{\#} N \end{aligned}$$

The proof script is:

```

1 CASE "M = {#}" THEN
2 CHOOSE "(M', m, M = M' + {#m#})" THEN
3 OBTAIN "m dvd (\prod_{i \in \#N} i)" THEN
4 CHOOSES ["(n, n \in \#N \wedge m dvd n)",
5         "(N', N = N' + {#n#})"] THEN
6 OBTAIN "m = n" THEN
7 OBTAIN "(\prod_{i \in \#M'} i) dvd (\prod_{i \in \#N'} i)" THEN
8 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:

$$\begin{aligned} \forall p \in \text{set_of } M. \text{ prime } p &\implies \forall p \in \text{set_of } N. \text{ prime } p \\ &\implies (\prod_{i \in \#M} i) = (\prod_{i \in \#N} i) \implies M = N \end{aligned}$$

with the proof script

```

1 OBTAIN "M \subseteq_{\#} N"

```

which tells the program to first show that M is a sub-multiset of N . After doing so, it suffices to show N is a sub-multiset of M . Both parts follow directly from the previous lemma.

3 Implementation

This section contains some notes on the implementation details of AUTO2, concluding with some examples of writing proof steps. Some parts of this section assume familiarity 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 $[T_1, \dots, T_n]$ 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 $f(t_1, \dots, t_n)$. If P is of the form $f(p_1, \dots, p_n)$, then the head matches of T against P is the result of matching the ordered list of terms $[t_1, \dots, t_n]$ against $[p_1, \dots, p_n]$ (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 .

3.3 More on box items

Each non-empty box contains a list of objects of type **box_item**, defined as

```
1 type box_item =
2   {id: box_id, sc: int, ty_str: string, tname: term list, prop: thm}
```

where

- **id** is the ID of the box containing the item.
- **sc** is the score (priority) of the item.
- **ty_str** is the string specifying the type of the item (**PROP**, **TERM**, **EQ**, etc).
- **tname** is a list of terms containing item information (for **PROP**, the statement of the proposition; for **EQ**, the two sides of the equality, etc).
- **prop** is the Isabelle theorem object justifying the item (if the item does not need to be justified due to its type, this is the trivial theorem **True**).

The objects contained in updates are of type **raw_item**, which is turned into **box_items** in the main loop. The datatype **raw_item** is defined as

```
1 datatype raw_item = Handler of term * retro_handler
2                   | Fact of string * term list * thm
```

The two possible types of raw items are:

- **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. In the main loop, this is added to the list of handlers in the appropriate box.
- **Fact** ($ty_str, tname, th$) represents a box item, containing type string, tname, and justifying theorem. 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.

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

3.4 Boxes, updates, and status

An object of type `box` contains all current information for one box ID. It is defined as:

```
1 type box = {vars: term list, assumes: term list, concls: term list,
2             handlers: (term * retro_handler) list,
3             items: (box_item * box_id list) ItemTab.table,
4             cbs: Update.resolve_callback list}
```

where

- `vars`, `assumes`, and `concls` contain the list of initial variables, assumptions, and conclusions in the box. These are present only for primitive boxes.
- `handlers` contains the list of retro-handlers.
- `items` is the list of box items. Each box item is associated to a list of box IDs under which it is shadowed.
- `cbs` is the list of callbacks to be called when the box is resolved.

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 primitive box, resolving a box, and shadowing an item). The type `update` is defined as the record

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

where

- `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:

```
1 type status = {
2   lat: BoxID.box_lattice,
3   boxes: Box.box Boxidtab.table,
4   queue: Updates_Heap.T,
5   rewrites: RewriteTable.rewrite_table,
6   ctxt: Proof.context
7 }
```

where

- `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.
- **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 (priority) 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 total size (**Term.size_of_term**) of the **tname** of the item. The idea is that we are less willing to add long-winded facts 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. If adding to a box that is the intersection of $n > 1$ primitive boxes outside the current focus, 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.

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 that performs matching with items and producing the new **raw_update** objects. The type is defined as

```
1 type proofstep_fn = box_item list -> rewrite_type -> status ->
2                   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

```

1 type pre_filter =
2   {ty_str: string, pre_filt: rewrite_table -> box_item -> bool}

```

It represents a filter to test whether it is possible for an item to serve as the first or second input to a proof step. Here if `ty_str` is the empty string, there is no constraint on type string. Otherwise the type string of the item must match `ty_str`. The pre-filter is used both as a prerequisite for invoking a proof step, and as a performance optimization: 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.

The type `proofstep` is defined as a record

```

1 type proofstep = {name: string, filt: pre_filter list,
2                  res_types: update_type list, func: proofstep_fn}

```

where

- `name` is a string that uniquely identifies the proof step.
- `filt` is the list of pre-filters on the inputs. Its length also specifies the number of input items the proof step takes.
- `res_types` is the list of possible types of raw updates in the returned value. In the main loop, proof steps that might shadow an item or resolve a box are tried first, since they may remove the new item added, so that trying the remaining proof steps become unnecessary.
- `func` is the actual proof step function.

In addition to defining the types, the structure `ProofStep` also provides some utility functions to help writing proof steps. The structure `ProofStepData` provides a second layer of utility functions, adding proof steps that directly apply a theorem using an even simpler notation. We discuss these two layers of functions in the next two subsections.

3.6.1 Proof steps applying a single theorem

Proof steps that directly apply a theorem can be written very simply, using functions in `ProofStepData`.

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 adds the conclusion of the theorem if it appears as a term in the current state.

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 outputting 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 outputting 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 “ \equiv ”) for rewriting. The rewriting is in the forward direction. The function `add_rewrite_rule.back` is used for rewriting in the backward direction. 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 2.9) 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 instantiations 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, 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:

```

1 val _ = Theory.setup (
2   add_gen_prfstep (
3     "dvd_transitive",
4     [WithFact @{term_pat "(?m::nat) dvd ?n"},
5       WithFact @{term_pat "(?n::nat) dvd ?p"},
6       GetFact (@{term_pat "(?m::nat) dvd ?p"},
7               @{thm dvd_transitive}),
8     Filter (neq_filter "m" "n"),
9     Filter (neq_filter "n" "p"),
10    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: $(?m::nat) \text{ dvd } ?n$ and $(?n::nat) \text{ dvd } ?p$. The third descriptor specifies the update produced: adding a new proposition $(?m::nat) \text{ dvd } ?p$. This is justified by the theorem with name `dvd_transitive`. The statement of this theorem is:

$$(?m::nat) \text{ dvd } ?n \implies ?n \text{ dvd } ?p \implies ?m \text{ dvd } ?p$$

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:

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

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

```
1 val _ = Theory.setup (
2   add_gen_prfstep (
3     "coprime_dvd_mult_nat",
4     [WithFact @{{term_pat "(?k::nat) dvd ?m * ?n"}},
5       WithGoal @{{term_pat "(?k::nat) dvd ?m"}},
6       GetGoal @{{term_pat "coprime (?k::nat) ?n"}},
7       @{{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:

$$\text{coprime } ?k \ ?n \implies ?k \text{ dvd } ?m * ?n \implies ?k \text{ dvd } ?m$$

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

$$?k \text{ dvd } ?m * ?n \implies \neg ?k \text{ dvd } ?m \implies \neg \text{coprime } ?k \ ?n$$

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:

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

3. (Creating a primitive box) The proof step `iff_intro1` is written as follows:

```
1 val _ = Theory.setup (
2   add_gen_prfstep (
3     "iff_intro1",
4     [WithGoal @{{term_pat "(?A::bool) = ?B"}},
5       CreateCase ([@{{term_pat "?A::bool"}}, [@{{term_pat "?B::bool"}]])])])
```

The first descriptor says the proof step should match pattern $?A \neq ?B$, where $?A$ and $?B$ are boolean variables. The second descriptor says when there is a match, a new primitive box should be created with initial assumption being the term substituted for $?A$, and initial conclusion being

the term substituted for ?B. Here `CreateCase` [*assums*] [*concls*] mean creating a primitive box with list of assumptions *assums* and list of goals *concls* (either of which can be empty).

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

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

where the theorem `Nat.less_not_refl` is $\neg((n::nat) < n)$. 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:

```
1 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:

```
1 val _ = Theory.setup (
2   add_prfstep_rewrite (
3     [WithTerm @{{term_pat "(?a::nat) ^ 1"}},
4       Filter (neq_filter "a" @{{term "1::nat"}})],
5     @{{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

```
1   ?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:

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

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

```
1 val _ = Theory.setup (
2   add_prfstep_two_stage (
3     "exists_n_dvd_n",
4     [WithGoal @{{term_pat "∃k. k dvd (?n::nat) ∧ ?A"}},
5       GetFact (@{{term_pat "(?n::nat) dvd ?n"}}, @{{thm n_dvd_n}})],
6     @{{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

$$\neg(\exists x. P\ x \wedge Q\ x) \implies P\ x \implies \neg(Q\ x)$$

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

7. (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:

```

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

The proof step `compare_consts` is written as follows:

```

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

```
7      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 to produce an update resolving the box.

3.6.3 Standard library of proof steps

Beyond these relatively simple examples, many proof steps are written in the standard library to implement more complicated behavior for induction, logic and (natural numbers) arithmetic. We briefly outline these functionalities:

- The induction proof steps are contained in structure `Induct.ProofSteps`. Simple induction, double induction, and strong induction are implemented using retro-handlers. Induction on inductively defined propositions and variables are implemented using box callbacks.
- `DISJ` and `DISJ_ACTIVE` items, and proof steps handling them, are implemented in structure `Logic.ProofSteps`. Functionalities include:
 - creating a `DISJ` or `DISJ_ACTIVE` from a disjunctive fact, conjunctive goal, forall fact, or exists goals.
 - Matching propositions to one term in the disjunction, hence reducing the item.
 - Adding boxes to start case analysis on `DISJ_ACTIVE` items.
 - Shadowing of redundant items.
- Inequalities on natural numbers up to constants: conversion of facts such as $a \leq b + 3$ and $a < b$, where a, b are terms of type `nat`, to a standard form of either $a \leq b + n$ or $a + n \leq b$, where n is a constant. Permit an item $a \leq b + n$ to justify any assumption $a \leq b + n'$ where $n' \geq n$ during matching in proof steps.