Auto2 Documentation (Draft)

Bohua Zhan

May 20, 2017

Contents

1 Introduction **2**

2 Description of the algorithm **2**
2.1 Box lattice . . . 4
2.2 Box items . . . 5
2.3 Proof steps . . . 6
2.4 Rewriting and matching . . . 9
2.5 Normalization (experimental) . . . 10
2.6 Retro-handlers and induction (experimental) . . . 10
2.6.1 Using an existence fact . . . 10
2.6.2 Induction . . . 11
2.7 Box callbacks . . . 12
2.8 Proof scripts . . . 12
2.9 Examples in number theory . . . 13
2.9.1 Example 1: `prime_imp_coprime_nat` . . . 13
2.9.2 Example 2: `not_prime_eq_prod_nat` . . . 14
2.9.3 Example 3: `prime_power_mult` . . . 15
2.9.4 Example 4: `prime_factor_nat` . . . 17
2.9.5 Example 5: Infinitude of prime numbers . . . 18
2.9.6 Example 6: Unique factorization theorem . . . 19

3 Implementation **21**
3.1 Subterms . . . 21
3.2 Rewrite table . . . 22
3.2.1 Simplification . . . 22
3.2.2 Adding new terms and equalities . . . 22
3.2.3 Matching . . . 23
3.2.4 Incremental matching . . . 24
3.2.5 Matching an equality pattern . . . 24
3.3 More on box items . . . 24
3.4 Boxes, updates, and status . . . 25
3.5 Scoring . . . 26

3.6 Adding proof steps 27
3.6.1 Direct application of a theorem 27
3.6.2 Constrained version of direct applications 28
3.6.3 Writing proof steps using descriptors 30
3.6.4 Custom proof steps 32
3.7 Adding item types 34
3.8 Adding new scripts 36
3.9 Standard library 37

1 Introduction

Auto2 is a theorem prover for higher-order classical logic written in Isabelle/ML, using Isabelle/HOL as the logical foundation. It is designed to combine a robust, saturation-based search procedure with the use of human-like heuristics. The intended application of auto2 is in assisting the formalization of usual developments in mathematics and computer science, by verifying those theorems or intermediate steps of a proof that humans would consider "routine".

The overall proof strategy of auto2 is a best-first search through the space of propositions that can be derived from a given list of assumptions, with the goal of eventually deriving a contradiction. The statement to be proved is first transformed into this contradiction form (for example, $[A, B] \Longrightarrow C$ is first transformed into $[A, B, \neg C] \Longrightarrow$ `False`). Human-like heuristics is used to determine what are the allowable steps in the best-first search. There are several elaborations to this basic picture, to naturally support reasoning with equalities, case analysis, and induction.

The prover is implemented as a tactic in Isabelle. It directly constructs the verified theorem object, hence its soundness is guaranteed by the Isabelle system. There is also an internal language of proof scripts that allows the user to provide intermediate steps for the proof of a theorem to auto2, in a way similar to (but currently independent from) Isar. The user may also extend the behavior of auto2 within Isabelle, for example to register the use of new theorems, or to implement more complex behavior.

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

2 Description of the algorithm

We begin with a high-level description of the algorithm, giving details in the succeeding sections.

As stated in the introduction, we begin by converting the statement to be proved into contradiction form. So we may assume that the goal is to derive a

contradiction from assumptions $A_1, \ldots, A_n$. Note that unlike most other classical theorem provers, we do not perform further normalizations to the statement. The previously proved theorems are not among the A_i's, instead they are encoded as proof steps, as we will explain later.

The algorithm maintains a list of *items*, which as a first approximation can be thought of as propositions derivable from the assumptions. More precisely, each item may be a proposition, or it may contain other information, in addition to or instead of a proposition. The list of items begins with the assumptions themselves. New items that may be added to the list are produced by *proof steps*, which are user-defined functions that, given as input one or two items, produce as output a list of new items that logically follow from the input items. The order in which the new items are added is governed by a scoring function: the new items are packaged into *updates*. Each update is given a non-negative integer score, and put into a priority queue. At each iteration, the update with the lowest score is pulled from the priority queue, and the items contained in the update are added to the list. This process continues until a contradiction (the proposition `False`) can be derived from the assumptions.

Each item in the list is assigned to a *box*, which describes what additional assumptions (if any) the item depends on. The algorithm maintains a list of primitive boxes, indexed by integers starting at zero. Each primitive box represents a list of assumptions. A box is given by a set of primitive boxes, representing the union of the initial assumptions of its member boxes. For an item that is purely a proposition, assigning it box b means the proposition is derivable from the assumptions in b. More generally, assigning an item to box b means the information represented by the item is available in the context of working under the assumptions of b. If a contradiction is derived in box b, we say b is *resolved*, and appropriate propositions (negations of the assumptions) are added to the parent boxes of b. New primitive boxes are produced by proof steps, and go through the priority queue just like items. Adding a primitive box can be thought of as starting a case analysis, as we will explain in Section 2.1.

Equality reasoning is handled using a data structure called the rewrite table. The rewrite table maintains the list of currently known equalities (without arbitrary variables). It provides a matching function that, given a pattern p, possibly containing arbitrary variables, and a concrete term t, produces the list of matches of t against p, up to rewriting t using the known equalities. The rewrite table automatically makes use of the transitive property of equality, as well as the congruence property (that is, $a_1 = b_1, \ldots, a_n = b_n$ means $f(a_1, \ldots, a_n) = f(b_1, \ldots, b_n)$). This is a well-known process called E-matching, used, for example, in most SMT (satisfiability-modulo-theories) solvers. The matching function is used as the first step of most proof steps. In addition to basic first-order matching up to equivalence, we also allow matching of certain higher-order patterns, and taking into account of associative-commutative (AC) functions. This is explained in more detail in Section 2.4.

Besides creating items and primitive boxes, a proof step may also *shadow* an item, removing it from consideration in the future. This is usually because the shadowed item is redundant with another item. We also separate out deriving

a contradiction in a box (resolving a box) as the fourth type of update from a proof step.

In summary, the state of the proof is made up of the list of primitive boxes, the list of items, the rewrite table, and the priority queue of future updates. Initially, everything is empty except for the priority queue, which contains one update, adding a primitive box (to be indexed 0) containing the initial assumptions of the statement to be proved. At each iteration of the loop, the update with the lowest score is pulled from the priority queue. Primitive boxes and items from the update are added to the state. For each new item added to the list, the following action is performed:

- (If the item is not an equality): All proof steps taking one item as input are invoked on the item. All proof steps taking two items as input are invoked on all pairs consisting of the new item and another item in the list. All resulting updates are added to the priority queue.
- (If the item is an equality): The equality is added to the rewrite table. All proof steps taking one input are invoked on all items containing (up to equivalence) either side of the equality. All proof steps taking two inputs are invoked on all pairs of items, of which at least one contains (up to equivalence) one side of the equality. All new updates (those that depend on the new equality) are added to the priority queue. This procedure is called *incremental matching*.

For efficiency, we may apply shadowing and resolving updates immediately, instead of adding them to the queue, since they always simplify the state.

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. Currently, we stop the algorithm after 2000 steps.

This completes a basic description of the algorithm. In the following subsections, we discuss various aspects of it in more detail.

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 composite box is a combination of a set of primitive boxes. Each primitive box also *inherits* from a composite box, taking all initial variables and assumptions from its parent. Primitive boxes are indexed by integers, so that a composite 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 composite 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 composite 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 $A \Longrightarrow$ `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 $AD \Longrightarrow$ `False` will resolve box $\{1\}$, and showing $BC \Longrightarrow$ `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 composite boxes i and j, we define the *intersection* of i and j to be the composite 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,...,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 $P_1 \vee \cdots \vee P_n$.

- `DISJ_ACTIVE` $[P_1,\ldots,P_n]$: same as `DISJ`, except when there are no schematic variables, the prover will attempt to derive a contradiction from $P_1 \vee \cdots \vee P_n$ 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 $[Q_1, ..., Q_i]$, 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 2.8).

- 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 direcly 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 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 ?a`^{0}, add new item `?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` $\neq$ `?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 `DISJ` [$\neg$`?A(?x), ?B(?x)`]. Associative property of disjunction is automatically applied. For example, given the item

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

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

9. (Resolving a disjunction) For any disjunction `DISJ` $[P_1, \ldots, P_n]$, 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 `DISJ` [$\neg$`?m dvd p, ?m = 1, ?m = p`], and item `n dvd p`, produce new item `DISJ` [`n = 1, n = p`].

10. (Creating a case from disjunction) Given active disjunction `DISJ_ACTIVE` $[P_1, \ldots, P_n]$, 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 $P_2, \ldots, P_n$.

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 `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 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 $A =$ `True` and $A \neq$ `False` by A. Replace $A =$ `False` and $A \neq$ `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 simplications are not added to the rewrite table.

2.6 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.

2.6.1 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

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

(`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.

2.6.2 Induction

Some forms of induction rules can be supported using retro-handlers (those that cannot are supported through box callbacks, see Section 2.7). One particularly common form is:

$$P(t_o) \Longrightarrow \forall t.(P'(t) \longrightarrow P(t)) \Longrightarrow \forall t.P(t), \tag{1}$$

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

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 $[P_1(n), ..., P_i(n)]$ 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 $P(n)$: $[P_1(n), ..., P_i(n)] \Longrightarrow \texttt{False}$. 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 actually a subcase of the above. The strong induction rule, for natural numbers or any set that is well ordered, is given by

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

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

2.7 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 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*, which is a theorem of the form $A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow B$, where each A_i is of the form

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

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 then. See the development of Hoare logic for examples.

2.8 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` $[s_1, \ldots, s_n]$ is equivalent to `CHOOSE` s_1 `THEN` $\cdots$ `THEN CHOOSE` s_n.

- `INDUCT` (n, `Arbitrary` m). Apply induction rule as described in Section 2.6. Let n_o be the value of n corresponding to the base case of the induction. Create box with assumption $n = n_o$, then add the induction assumption for n, holding m 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 2.6) or general variable induction (Section 2.7). 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.

2.9 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.

2.9.1 Example 1: `prime_imp_coprime_nat`

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

$$\texttt{[prime p, } \neg \texttt{ p dvd n]} \Longrightarrow \texttt{coprime p n}$$

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`, `¬ p dvd n`, and `gcd p n ≠ 1`.

1. From `TERM prime p`, add `prime p = (1 < p ∧ ∀m. m dvd p ⟶ m = 1 ∨ m = p)` to box 0.
2. Using equality from the previous step, `prime p` matches pattern `?A ∧ ?B`, which adds `1 < p` and `∀m. m dvd p ⟶ m = 1 ∨ m = p` to box 0.
3. From `∀m. m dvd p ⟶ m = 1 ∨ m = p`, add `DISJ [¬ ?m dvd p, ?m = 1, ?m = p]`.
4. From `TERM gcd p n`, add basic properties of `gcd`: `gcd p n dvd p` and `gcd p n dvd n`.
5. The negation of `gcd p n dvd p` matches the first term of disjunction `DISJ [¬ ?m dvd p, ?m = 1, ?m = p]`. This produces `DISJ [gcd p n = 1, gcd p n = p]`.
6. The negation of `gcd p n ≠ 1` matches the first term of `DISJ [gcd p n = 1, gcd p n = p]`. This results in item `gcd p n = p`.
7. However, with `gcd p n = p`, there is a contradiction between assumption `¬ p dvd n` and derived fact `gcd p n dvd n`. This proves the theorem.

2.9.2 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, ¬ 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`, and `¬ ∃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 the pattern `¬(?A ∧ ?B)`. Applying de Morgan's law, we get item `DISJ_ACTIVE [¬ 1 < n, ¬ ∀m. m dvd n ⟶ m = 1 ∨ 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 `¬ ∀m. m dvd n ⟶ m = 1 ∨ m = n`.
4. `TERM ¬ ∀m. m dvd n ⟶ m = 1 ∨ m = n` matches pattern `TERM ¬ ∀x. ?A(x)`, adding an equality rewriting it to `∃m. ¬(m dvd n ⟶ m = 1 ∨ m = n)`.

5. The result of previous step matches pattern ∃x. `?A(x)`. Using the existence fact, we may add a new variable `m` and fact ¬`(m dvd n` ⟶ `m = 1 ∨ m = n)`.

6. From the previous step, 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. Item `m dvd n` from the previous step rewrites to ∃`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` ≤ `m` ≤ `n` and `1` ≤ `k` ≤ `n`.

9. From the assumption ¬ ∃`m k. n = m * k` ∧ `1 < m` ∧ `m < n` ∧ `1 < k` ∧ `k < n`, add `DISJ_ACTIVE` [`n` ≠ `?m * ?k, ¬ 1 < ?m, ¬ ?m < n, ¬ 1 < ?k, ¬ ?k < n`]. The first term of this item matches the negation of `n = m * k`, giving `DISJ_ACTIVE` [¬ `1 < m, ¬ m < n, ¬ 1 < k, ¬ k < n`].

10. From the previous `DISJ_ACTIVE` item, add primitive box 1 under {0} with assumption ¬ `1 < m`.

11. In box {1}, it suffices to show `1 < m`, and `m` ≠ `1` is known in box {0}, so it suffices to show `1` ≤ `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` [¬ `m < n, ¬ 1 < k, ¬ k < n`] are added to box {0}.

12. Primitive box 2 is created under {0} with assumption ¬ `m < n`. This is resolved similarly as before, using `m` ≠ `n` from Step 6 and `m` ≤ `n` from Step 8. This adds `m < n` and new item `DISJ_ACTIVE` [¬ `1 < k, ¬ k < n`] to box {0}.

13. Primitive box 3 is created under {0} with assumption ¬ `1 < k`. Since `1` ≤ `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` ≠ `n` from Step 6. This resolves box {3}, and adds `1 < k` and ¬ `k < n` as facts to {0}.

15. Since `k` ≤ `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` ≠ `1` from Step 6. This resolves box {0} and proves the theorem.

2.9.3 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, indicating an induction on k, with x and y held arbitrary.

```
CASE "k = 0" THEN INDUCT ("k", 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 `[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`). Then we have `DISJ_ACTIVE` [`p dvd x, p 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 ∃`i. x = p`^{i} in box {2} generates item `DISJ` [¬ `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 ∃`i. x = p`^{i} to box {3} and ¬ `p dvd x` to box {2}. Then goal ∃`j. y = p`^{j} is added to box {3}.

14. In box {3}, goal ∃`j. y = p`^{j} generates the item `DISJ` [¬ `y = p`$^{?j}$]. Since `y = p`^{j} (from Step 10) matches the left side of this implication, box {3} is resolved. Facts ¬ `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.

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

2.9.4 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

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

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 ∀`m<n. m` ≠ `1` ⟶ `(`∃`p. prime p` ∧ `p dvd m)` to box {0}. The rest of the proof proceeds as follows:

1. Since `n dvd n`, we obtain the fact ¬ `prime n`. This gives ¬`(1 < n)` ∨ ¬`(`∀`m. m dvd n` ⟶ `m = 1` ∨ `m = n)`.

2. In the first case, we have ¬`(1 < n)` or `n` ≤ `1`. Since `n` ≠ `1`, we get `n < 1` and then `n = 0`. Since `p dvd 0` holds for any `p`, it suffices to show ∃`p. prime p`. But this is a known result.

3. In the second case, instantiate with variable `m` and proposition ¬`(m dvd n` ⟶ `m = 1` ∨ `m = n)`. This becomes a conjunction of facts `m dvd n`, `m` ≠ `1`, and `m` ≠ `n`. From `m dvd n` we get `m` ≤ `n`, which when combined with `m` ≠ `n` gives `m < n`. From the strong induction hypothesis, we get ∃`p. prime`

`p` $\wedge$ `p dvd m`. Instantiate with variable `p` and propositions `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 ∃`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.

2.9.5 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

∃`p`. `prime p` $\wedge$ `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` $\wedge$ `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` $\neq$ `1`. Alternatively, we have fact `fact n + 1 = 1`. This becomes `fact n = 0`, which contradicts the fact `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 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 dvd fact n`. By backward reasoning with theorem `[1` $\leq$ `p, p` $\leq$ `n]` $\Longrightarrow$ `p dvd fact n`, it suffices to show `1` $\leq$ `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` $\leq$ `n`.

3. The new fact ¬ `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 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

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

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.

2.9.6 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:

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

Here `M` ranges over finite multisets of natural numbers. The term `set_of M` is the set of elements in `M` (forgetting multiplicity). The notation ∏`i`∈#`M. i` means "the product of `i` when `i` ranges over `M`, counting multiplicity".

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

```
STRONG_INDUCT ("n", []) THEN
CASE "n = 1" WITH OBTAIN "n = (∏i∈#{#}. i)" THEN
CASE "prime n" WITH OBTAIN "n = (∏i∈#{#n#}. i)" THEN
CHOOSES ["(m, k, n = m * k ∧ 1 < m ∧ m < n ∧ 1 < k ∧ k < n)",
        "(M, (∀p∈set_mset M. prime p) ∧ m = (∏i∈#M. i))",
        "(K, (∀p∈set_mset K. prime p) ∧ k = (∏i∈#K. i))"] THEN
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 = ∏i∈#?M. i`, obtaining a match with `?M := {#}`. This gives the goal `∀p ∈ set_of {#}. prime p`, which is easy to prove. This adds `n ≠ 1` and then `n > 1` as facts.

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`. This adds `¬ 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 `¬ 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 = (∏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:

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

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:

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

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:

```
∀p∈set_of M. prime p  ⟹  ∀p∈set_of N. prime p
                      ⟹  (∏i∈#M. i) = (∏i∈#N. i) ⟹ M = N
```

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, 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 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.

3.3 More on box items

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

```
type box_item =
    {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 justifyed 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_item`s in the main loop. The datatype `raw_item` is defined as

```
datatype raw_item = Handler of term * retro_handler
                  | 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:

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

where

- `vars`, `assums`, 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

```
type update = {sc: int, prfstep_name: string,
               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:

```
type status = {
  lat: BoxID.box_lattice,
  boxes: Box.box Boxidtab.table,
  queue: Updates_Heap.T,
  rewrites: RewriteTable.rewrite_table,
  ctxt: Proof.context
}
```

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 Adding proof steps

Adding new proof steps is the most basic way to extend `auto2`. In the following subsections, we discuss different ways of adding proof steps, in order of increasing complexity.

3.6.1 Direct application of a theorem

The simplest kind of proof steps are those that directly applies a theorem. There are two ways to quickly add such a proof step: via an Isabelle attribute or by calling a setup function. Isabelle attributes can be used for theorems stated in the theory. They are specified in square brackets after the name of the theorem. For example, to add theorem `XYZ` as a forward proof step, one writes

```
theorem XYZ [forward]: ...
```

Setup functions can be used for all theorems. For example, if we want to define function `foo` and add the automatically generated theorem `foo_def` as a rewrite rule, we can write:

```
definition foo: ...
setup {* add_rewrite_rule @{thm foo_def} *}
```

The list of basic proof steps, together with the corresponding attributes and setup functions, are as follows:

- Forward proof step (attribute `forward`, setup function `add_forward_prfstep`): The theorem must have at least one assumption. If there are one or two assumptions in the theorem, match the assumptions against the same number of items, and output the conclusion of the theorem. If there are more than two assumptions, match the first two assumptions, and output the remaining assumptions and the conclusion as an implication (to be converted into a passive disjunction). Every schematic variable appearing in the output portion of the theorem must appear in the matched portion of the theorem.

- Backward proof step (attribute `backward`, setup function `add_backward_prfstep`): The theorem must have exactly one assumption, in the form $P \Longrightarrow Q$. The proof step matches $\neg Q$ against one item, and outputs $\neg P$. Any schematic variable appearing in P must appear in Q.

- First backward proof step (attribute `backward1`, setup function `add_backward1_prfstep`): The theorem must have exactly two assumptions, in the form $P \implies Q \implies R$. The proof step matches Q and $\neg R$ against one item, and outputs $\neg P$. Any schematic variable appearing in P must appear in either Q or R.

- Second backward proof step (attribute `backward2`, setup function `add_backward2_prfstep`): Same as `backward1`, except using the second assumption instead of the first assumption.

- Resolve proof step (attribute `resolve`, setup function `add_resolve_prfstep`): The theorem must have 0 or 1 assumptions. Match the assumption (if any) and the negation of the conclusion against one or two items, and output a contradiction for any match.

- Rewrite proof step (attribute `rewrite`, setup function `add_rewrite_rule`: The conclusion of the theorem must be an equality. If the theorem has 0 or 1 assumptions, match the assumptions and the left side of the conclusion with one or two items, output the equality for any match. If the theorem has more than 1 assumptions, match the first assumption and the left side of the conclusion, output the remaining assumptions and the conclusion as an implication (to be converted into a passive disjunction). Every schematic variable appearing in the output portion of the theorem must appear in the matched portion of the theorem.

- Backward rewrite proof step (attribute `rewrite_back`, setup function `add_rewrite_rule_back`: Same as `rewrite`, except use the right side of the equality instead of the left side.

- Bidirectional rewrite proof step (attribute `rewrite_bidir`, setup function `add_rewrite_rule_bidir`: Same as adding both `rewrite` and `rewrite_back`.

3.6.2 Constrained version of direct applications

The next simplest kind of proof steps are those that applies a theorem under some constraints. There are two kinds of constraints. The first kind specifies a certain term that must exist for the theorem to be applied. The second kind specifies a filter on the instantiation of the schematic variables. These two are fundamentally different in that each term constraint requires matching an item, hence in the forward and rewrite proof steps, one less assumption can be matched. The backward proof step can take at most one term constraint, and the backward$_1$ and backward$_2$ proof steps can take none (there is no reason to impose constraints of any kind on resolve proof steps). The filter constraints do not require matching additional items, so there is no limit on their numbers.

Proof steps with constraints can be added using a setup function. The name of the setup function is obtained from the original name by adding suffix `_cond`. The list of constraints is given as the second argument to the setup function.

The constraint `with_term` t means t must be a term in the current state. Here t may contain any schematic variables appearing in the matched portion of the theorem.

The constraint `with_cond` $?a \neq p$, where p is a term that may contain schematic variables, specifies that the instantiation of $?a$ must not match the pattern p. Any schematic variables appearing in p that also appear in the matched portion of the theorem will be instantiated, otherwise they will remain as schematic variables. The matching is up to rewriting using known equalities. In particular, `with_cond` $?a \neq ?b$ means the instantiation of $?a$ and $?b$ must be non-equivalent.

The constraint `with_filt` f is the most general kind. Here f is a function of type `prfstep_filter`, which is defined as:

```
type prfstep_filter = rewrite_table -> id_inst -> bool
```

The arguments are the current rewrite table and the instantiation, and the return value is whether the instantiation passes the filter. A list of common filters are defined in `proofstep.ML`:

- `id_filter` f: The box ID of the instantiation must satisfy function f (of type `box_id -> bool`).
- `not_numc_filter` s: The schematic variable $?s$ must *not* be equivalent to a numeric constant.
- `order_filter` s_1 s_2: The instantiation of $?s_1$ must be less than or equal to the instantiation of $?s_2$, under Isabelle's term ordering.
- `size1_filter` s: The instantiation of $?s$ must simplify to a term of size one (constant or free variable).
- `not_type_filter` s ty: The type of $?s$ must not equal ty.
- `unique_free_filter` s: The instantiation of s must be a free variable. Moreover, it must be the only free variable of the type of s in the current context.
- `ac_atomic_filter` nm s: Given nm is the name of an AC function $\star$, require $?s$ is atomic with respect to $\star$ (that is, it is not directly in the form $a \star b$).
- `canonical_split_filter` nm s_1 s_2: Given nm the name of an AC function $\star$, require that $?s_1 \star ?s_2$ is a canonical splitting. If $\star$ associates to the left (the usual case), this means $?s_2$ is atomic with respect to $\star$, and $?s_1$ consists of terms all of which are less than or equal to $?s_2$.

Example With theorem

```
Nat.dvd.order.trans: ?a dvd ?b ⟹ ?b dvd ?c ⟹ ?a dvd ?c
```

the setup

```
add_forward_prfstep_cond @{thm Nat.dvd.order.trans}
  (with_conds ["?a ≠ ?b", "?b ≠ ?c", "?a ≠ ?c"])
```

adds a proof step that matches two items against patterns `?a dvd ?b` and `?b dvd ?c`, and outputs the conclusion `?a dvd ?c`, provided that no two of `?a`, `?b` and `?c` are known to be equivalent.

Example With theorem

```
prime_dvd_mult_nat: prime p ⟹ p dvd m * n ⟹ p dvd m ∨ p dvd n
```

the setup

```
add_forward_prfstep_cond @{thm prime_dvd_mult_nat}
  (with_filts [canonical_split_filter @{const_name times} "m" "n"] @
   with_conds ["?m ≠ ?p", "?n ≠ ?p", "?m ≠ ?p * ?m'", "?n ≠ ?p * ?n'"])
```

adds a proof step that matches two items against patterns `prime ?p` and `?p dvd ?m * ?n`, and returns the disjunction `?p dvd ?m ∨ ?p dvd ?n`. The first filter stipulates that out of the many matches of `?m * ?n` against a term a_1 `*` ... `*` a_k (due to the AC property of multiplication), only one, with `?m =` a_1 `*` ... `*` a_{k-1} and `?n =` a_k, will contribute. The remaining filters stipulate that `?p` must not be a factor in either `?m` or `?n` (otherwise either `?p dvd ?m` or `?p dvd ?n` will be trivially true).

3.6.3 Writing proof steps using descriptors

At the next level, we consider writing proof steps using descriptors. The datatype `prfstep_descriptor` is defined as:

```
datatype prfstep_descriptor = WithProp of term
                            | WithItem of string * term
                            | GetFact of term * thm
                            | ShadowFirst | ShadowSecond
                            | CreateCase of term list * term list
                            | Filter of prfstep_filter
```

Each descriptor specifies either an input item, an output, or a filter for the proof step. The meaning of the different types of descriptors are as follows:

- `WithFact` p, where p is a pattern: match an item for the proposition p.
- `WithItem (`*ty_str*`,` p`)`, where *ty_str* is a string and p is a pattern: match against patterns with given *ty_str*, using the custom matching function for that type. In particular, `WithItem (TERM,` p`)` means matching p against a term.

- `GetFact (`p`, `th`)`: produce a conclusion p using theorem th.
- `ShadowFirst` (resp. `ShadowSecond`): shadow the first (resp. the second) item in the input.
- `CreateCase (`$assums$`, `$concls$`)`: create a new box with the given assumptions and conclusions.
- `Filter` f: specify the given function as a filter.

There are also two functions defining short-hands:

- `WithGoal` p: equivalent to `WithFact` $\neg p$.
- `WithTerm` p: equivalent to `WithItem (TERM,` p`)`.

When specifying proof steps using descriptors, one should be careful to specify the types of terms. If the type is arbitrary, it should be a schematic (not free) type variable.

The simplest kind of writing proof steps using descriptors is with `add_gen_prfstep` (there is a corresponding function `gen_prfstep` that creates the proof step, to be added at any convenient time).

```
val add_gen_prfstep: string * prfstep_descriptor list -> theory -> theory
```

The arguments to `add_gen_prfstep` is the pair $(name, descs)$, where $name$ is the name of the proof step (which must be unique), and $descs$ is the list of descriptors specifying the proof step.

Example The following setup

```
add_gen_prfstep ("iff_intro1",
  [WithGoal @{term_pat "(?A::bool) = ?B"},
   CreateCase ([@{term_pat "?A::bool"}], [])])
```

adds a proof step named `iff_intro1` which, on seeing a goal to prove `?A = ?B` (`?A` if and only if `?B`), creates a case assuming `?A`. This corresponds to proving an if and only if statement $A = B$ by first proving the direction $A \longrightarrow B$.

One can also write proof steps that applies conversions. A conversion in Isabelle is a function of type $cterm- > thm$ that, given a certified term as input, returns a Pure equality theorem from that term to an equivalent term. The function to add a conversion as a proof step is:

```
val add_prfstep_conv:
    string * prfstep_descriptor list * conv -> theory -> theory
```

(the corresponding function `prfstep_conv` creates a proof step to be added at any time). The list of descriptors should contain exactly one term pattern. The proof step matches that term pattern, and applies the conversion to the matched term. If the conversion fails, the proof step throws an error. If the conversion returns an identity equality (or an equality that is already known) the proof step does not create an update.

Example Suppose `nat_fold_conv` is a conversion that performs arithmetic operations on natural numbers (for example, will return $2 + 3 = 5$ for input $2 + 3$. The setup

```
add_prfstep_conv
  ("eval_plus_consts",
   [WithTerm @{term_pat "(?NUMC1::nat) + ?NUMC2"},
    Filter (order_filter_n ("NUMC", 1) ("NUMC", 2)),
    Filter (fn _ => fn (_, inst) =>
              lookup_numc1 inst > 0 andalso lookup_numc2 inst > 0)],
   nat_fold_conv)
```

finds terms that are sums of two constant natural numbers, neither of which is zero, and performs the addition.

It is also possible to create proof steps for conversions that takes the Isabelle context as an argument, using the `prfstep_pre_conv` function:

```
val add_prfstep_pre_conv: string * prfstep_descriptor list *
                          (Proof.context -> conv) -> theory -> theory
```

3.6.4 Custom proof steps

Writing proof steps directly in ML is the last resort, when any of the above will not suffice. There are two levels of abstraction for writing such proof steps. First, the function `add_prfstep_custom` abstracts away the portion of the proof step for matching items, so that the user only has to write how to obtain the update from the matched items:

```
val add_prfstep_custom:
      (string * prfstep_descriptor list * Update.update_type list *
       (id_inst_ths -> box_item list -> status -> Update.raw_update list)) ->
      theory -> theory
```

Here the third argument specofies the possible types of updates the proof step may return. The last argument specifies the function for obtaining updates. The arguments to that function are: the instantiation and theorems from matched items, the matched items themselves, and the current status.

Example We give the example for checking inequalities between constant natural numbers. The following setup

```
add_prfstep_custom
  ("compare_consts",
   [WithFact @{term_pat "(?NUMC1::nat) = ?NUMC2"},
    Filter (fn _ => fn (_, inst) =>
              lookup_numc1 inst <> lookup_numc2 inst)],
   [Update.RESOLVE_BOX],
   fn ((id, _), ths) => fn _ => fn {ctxt, ...} =>
      [Update.thm_update (id, contra_by_arith ctxt ths)])
```

matches a fact $m = n$, where m and n are constant natural numbers, and returns a contradiction if m and n are in fact not equal. The contradiction is obtained by calling the `arith` tactic on the statement $m = n \implies$ `False`.

If even `prfstep_custom` is insufficient, one must write the proof step directly. The type of proof steps is defined as:

```
datatype match_arg = PropMatch of term | TypedMatch of string * term

datatype proofstep_fn
  = OneStep of rewrite_type -> status -> box_item -> raw_update list
  | TwoStep of rewrite_type -> status -> box_item -> box_item -> raw_update list

type proofstep = {name: string, args: match_arg list,
                  res_types: update_type list, func: proofstep_fn}
```

Here `proofstep_fn` is the type for the core function of the proof step. Its two options are for proof steps matching one or two items. The type `proofstep` specifies additional information about a proof step, including its name, the patterns the items need to match for the proof step to be invoked, and the possible types of updates.

Example We give as example one of the core proof steps: starting a case analysis from a disjunction. The code is:

```
fun disj_create_case_fn rtype _ {id, tname, ty_str, ...} =
    if not (RewriteTable.is_single_rtype rtype) then [] else
    if has_vars tname then [] else
    let
      val _ = assert (ty_str = TY_DISJ_ACTIVE) "disj_concl_active: wrong type."
      val subs = HOLogic.strip_tuple tname
      val assum = HOLogic.mk_Trueprop (hd subs)
    in
      if length subs = 1 then []
      else [Update.AddBoxes {
               id = id, inits = [Update.InitAssum assum], cbs = []}]
    end

val disj_create_case_prfstep =
    {name = "disj_create_case",
     args = [TypedMatch (TY_DISJ_ACTIVE, @{term_pat "?C | ?D"})],
     res_types = [Update.ADD_BOXES],
     func = OneStep disj_create_case_fn}
```

First, let's examine the proof step function. It first checks that the `rtype` does not specify incremental matching. This is because no matching is done within the proof step, so it should not create anything during incremental matching. Next, it checks that the item does not contain schematic variables – no case analysis is created from items with schematic variables. After these two checks,

it takes the first term A in the disjunction and, provided the item is the disjunction of at least two terms, creates a case analysis with assumption A. In the second part, we add other information about the proof step: its name, the fact that it matches only `DISJ_ACTIVE` items, and that it produces updates that add new boxes.

3.7 Adding item types

In this section, we discuss the detailed mechanics about adding new item types. These provide powerful tools for encoding more complex behavior into proof steps.

Each item type is identified by a string. For example, the item type for propositions is "`PROP`". The item type for terms is "`TERM`". The item types for disjunctions are "`DISJ`" and "`DISJ_ACTIVE`".

The type `raw_item` are objects contained in updates that add new items. The type is defined as:

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

We will discuss the `Handler` objects in the section on retro handlers. For now, we will focus on the usual items, which are the `Fact` objects. Each fact object is constructed using a string, which is the name of the item type, an Isabelle term, which is the tname of the item, and a justifying theorem.

Each item has a printing function, which is used when printing trace information. The printing function is of type `item_output`, defined as:

```
type item_output = Proof.context -> term * thm -> string
```

To register an item type, use the setup function `add_item_type`:

```
val add_item_type: string * item_output option -> theory -> theory
```

The two arguments are the (unique) name of the item type, and an optional `item_output` function. If none is given, the default one (printing the tname of the item) will be used.

Next, we consider input and output of custom item types in proof steps. The output part is simple: construct the `raw_item` using the `Fact` constructor, providing the name of the item type, the tname, and the theorem for the item.

For input into proof steps, one need to write item matchers. The type `item_matcher` is given by:

```
type item_matcher = {
  pre_match: term -> box_item -> rewrite_table -> bool,
  match: term -> box_item -> rewrite_table -> id_inst -> id_inst_th list
}
```

Each item matcher consists of two functions. The `pre_match` function is invoked to filter the list of items when applying proof steps with two inputs. The function should be defined so that `pre_match` *pat item tbl* returns whether it is possible

for the item to match the pattern, with the given rewrite table. The match function performs the actual matching. It is given the same inputs, together with an existing instantiation σ. The function should return the list of extensions to σ, together with a theorem, derived from the justifying theorem of the item.

There are two kinds of input matchers, corresponding to the `WithFact` and `WithItem` descriptor for a proof step. The `WithFact` descriptor invokes a prop matcher, while the `WithItem` descriptor invokes a typed matcher. The functions adding these two input matchers are:

```
val add_prop_matcher: string * item_matcher -> theory -> theory
val add_typed_matcher: string * item_matcher -> theory -> theory
```

We give as example the term type. The type is given the type string "`TERM`":

```
val TY_TERM = "TERM"
```

The setup for adding the item type is:

```
add_item_type (TY_ITEM, NONE)
```

Here `NONE` indicates that the printing function for the item is just printing its tname.

Next, the typed matcher for `TERM` is:

```
val term_typed_matcher =
    let
      fun pre_match pat {id, tname, ...} tbl =
          length (fo_table_match_head id tbl (pat, tname)) > 0

      fun match pat {tname, ...} tbl (id, inst) =
          let
            val insts' =
                RewriteTable.fo_rewrite_match_head tbl (pat, tname) (id, inst)
            fun process_inst (inst, _) = (inst, true_th)
          in
            map process_inst insts'
          end
    in
      {pre_match = pre_match, match = match}
    end
```

This is invoked on pattern p when `WithItem (TERM, p)` is a descriptor. The `pre_match` function says for the item to match, the tname of the item must match p. The match function then performs the actual match. Note the returned function is `true_th` (the trivial theorem `True`), because no justifying theorem is needed for a term.

The prop matcher for `TERM` is:

```
val term_prop_matcher =
    let
```

```
    fun pre_match pat {id, tname, ...} tbl =
        case pat of
            Const ("HOL.eq", _) $ lhs $ _ =>
            length (fo_table_match id tbl (lhs, tname)) > 0
          | _ => false

    fun match pat {tname = u, ...} tbl (id, inst) =
        if fastype_of pat <> boolT orelse not (is_eq_term pat) then [] else
        let
          val (lhs, rhs) = pat |> HOLogic.dest_eq
          val insts' =
              RewriteTable.fo_rewrite_match_list
                  tbl [(true, (lhs, u)), (false, (rhs, u))] (id, inst)
          fun process_inst (inst, ths) =
              let
                (* th1: lhs(env) == u, th2: rhs(env) == u. *)
                val (th1, th2) = the_pair ths
              in
                (inst, to_obj_eq (transitive_list [th1, meta_sym th2]))
              end
        in
          map process_inst insts'
        end
  in
    {pre_match = pre_match, match = match}
  end
```

This is the prop matcher for obtaining equalities from term items. Given a term t and an equality pattern $p = q$, we say t matches $p = q$ if t matches p at head (that is, only rewriting subterms of t), and matches q generally (allowing rewriting t as a whole). In the `pre_match` function, we check that the pattern to be matched is indeed an equality, and that the term matches the left side of the equality. In the `match` function, we perform the matching as described. The equality theorem is returned: given instantiation σ, we form the equality $p(\sigma) = u = q(\sigma)$.

For further examples of defining new item types, see types `DISJ` and `DISJ_ACTIVE` in `logic_steps.ML`, types `NAT_ORDER` in `order.ML`, and `STRICT_LINORDER` in `Arrows_Ex/strict_lin_order.ML`.

3.8 Adding new scripts

The data type script is defined as:

```
datatype script
  = Script_Task of {inits: Update.init_info list, subs: script list,
                    cb: (box_id * thm) * status -> Update.raw_update list,
                    end_vars: term list}
```

Each script is specified by:

- `inits:` The initial variables and assumptions of the box.
- `subs:` The nested scripts. This is usually left empty when defining new scripts, to be inserted using the `WITH` form.
- `cb:` The callback function, specifying what to do when the new box is resolved. The function takes as argument the ID and resolving theorem of the box, and the status when the box is resolved, and returns the list of updates.
- `end_vars:` The list of fixed variables: these variables are reserved for the callback function. They will not appear anywhere else in the proof.

For examples of defined scripts, see the definition of `CHOOSE` in `logic_steps.ML` and the induction scripts in `induction.ML`.

3.9 Standard library

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.