AUTO2 Documentation (draft)

Bohua Zhan

March 16, 2015

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 usual developments in mathematics and computer science. The overall proof strategy is a best-first search through statements that can be derived from a given list of assumptions, with the goal of eventually deriving a contradiction. The theorem to be proved is always first transformed into this form (for example, $[A, B] \Longrightarrow C$ is first transformed into $[A, B, \neg\ C] \Longrightarrow$ `False`). In implementing this strategy, emphasis is placed on being able to naturally support rewriting, proof by case analysis, and induction. 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 three examples from elementary number theory. No knowledge of Isabelle is assumed in this part. In the second part, we describe some implementation details.

1 Overview of algorithm

1.1 Box lattice

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

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

For a more complicated example, suppose $A \vee B$ and $C \vee D$ are present in box $\{0\}$, and the proof is to proceed by checking each of the four cases AC, AD, BC, and BD. To do so, we create primitive boxes 1 and 2, both inheriting from $\{0\}$, with A and C respectively as the additional assumption. Once a contradiction is derived from AC, the box $\{1, 2\}$ is resolved. This puts propositions $\neg A$ in box $\{2\}$, and $\neg C$ in box $\{1\}$, from which we can derive B in box $\{2\}$, and D in box $\{1\}$. Showing $AD \Longrightarrow \texttt{False}$ will resolve box $\{1\}$, and showing $BC \Longrightarrow \texttt{False}$ will resolve box $\{2\}$. This will then put $\neg A$ and $\neg C$ in box $\{0\}$, which allows us to put 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 can define the *intersection* of i and j to be the general box containing exactly the assumptions in i and j. This is formed by merging the two lists of primitive boxes describing i and j, and removing any redundancies. We say i is the *eq-ancestor* of j, or j is the *eq-descendent* of i, if the intersection of i and j equals j.

1.2 Box items

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

Each item is assigned an integer called *score*, which directs the best-first search. From the point of view of best-first search, a lower score means it is more attractive to use this item to derive further items. In the current implementation, scores are assigned cumulatively: the initial assumptions in primitive box 0 are given score 0. Any time an item P is derived from a list of items $[Q_1, ..., Q_i]$, the score of P equals the maximum of the scores of Q_i, plus an additional value that depends on P (we can also let it depend on how P is derived, although this is not yet implemented). The score of initial assumptions in a primitive box other than 0 is calculated similarly from the scores of items

that causes this box to be created. In this view, the score also measures the distance of any given item from the initial assumptions at 0.

1.3 Proof steps

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

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

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 any item with proposition $\neg P$ 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 prove 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 is more naturally expressed as a list of properties. Here when gcd is encountered, we introduce part of the usual definition for gcd.

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

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

8. (Adding an implication) If an item matches ∀`x. ?A(x) ⟶ ?B(x)`, add new item with implication `?A(?x) ⟹ ?B(?x);`. For example, on seeing ∀`m. m dvd p ⟶ m = 1 ∨ m = p`, add a new implication `?m dvd p ⟹ ?m = 1 ∨ ?m = p`.

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

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

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

In most cases, a proof step is justified by a single theorem and simply applies that theorem. The additional information that a proof step provides is how that theorem is to be used: whether in the forward or backward direction, and whether there are restrictions on the values of arbitrary variables. In this sense, proof steps can be considered as information on how to use the available theorems to reason within a particular mathematical theory. Ideally, a proof step is added right after the theorem justifying it is proved, and will be available in all subsequent proofs. This means no extra "hints" 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. See the section on proof steps in the implementation part of this documentation for some examples.

1.4 Rewriting and matching

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

The interface provided by the rewrite table is relatively simple. As input, one can add a new equality to a rewrite table under a particular box (the rewrite

table is designed to handle equalities under different boxes). As output, the rewrite table provides two services: it finds the simplest known form for a given expression, and it produces all matches of a pattern against an expression, after rewriting the expression using the list of known equalities. Only first order matching (not unification or higher order matching) is available, although the matcher can "go inside" $\forall$ and $\exists$ statements, so that proof steps in examples 8 and 10 above work as intended. Using the simplify function, we can also check whether any two expressions are known to be equivalent.

As an example, suppose `prime p` = `1 < p` $\wedge$ $\forall$`m. m dvd p` $\longrightarrow$ `m = 1` $\vee$ `m = p` is added to the rewrite table. Then the term $\neg$ `prime p` matches the pattern $\neg$ `(?A` $\wedge$ `?B)`, with `?A := 1 < p` and `?B :=` $\forall$`m. m dvd p` $\longrightarrow$ `m = 1` $\vee$ `m = p`. This allows de Morgan's law to be applied to $\neg$ `prime p`. The ability to match modulo 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`.

1.5 The main loop

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

1. The new item is matched against all proof steps. For proof steps that

match two items, it is matched while paired with each of the other items in the state. All new updates created are added to the priority queue.

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

This procedure continues, until box $\{0\}$ is resolved, or until certain time-out conditions are reached. One way to measure the amount of work done is 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.

1.6 Retro-handlers and induction

Sometimes it can be convenient to add propositions that do not logically follow from the assumptions, but can be assumed for the purpose of the proof, because any proof of contradiction using it can be transformed to one that does not. The most immediate example is using an existence fact. If $\exists$`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 did not appear anywhere else in the proof. While `P x` does not logically follow from the assumptions (which do not even mention `x`), any proof using `P x` and with a conclusion that does not involve `x` can be transformed into a proof that does not use `P x`, by applying the theorem $\exists$`x. ?P x` $\Longrightarrow$ `(`$\bigwedge$`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 depends on `P x`, a *retro-handler* is invoked to retroactively remove that 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 $\bigwedge$`x. (P x` $\Longrightarrow$ `False)` will be available to the parent box, so that a contradiction can be derived in the parent box by applying `exE`. The approach currently used, involving retro-handlers, is chosen mainly to reduce the number of primitive boxes needed in a proof.

A second use of retro-handlers is to support induction. Currently only inductions on natural numbers are implemented, and for simplicity we only discuss this example. To perform a simple induction on n, first a primitive box is created for the $n = 0$ case. Induction begins when the $n = 0$ case is resolved adding $n \neq 0$ to the parent box. Let $[P_1(n), ..., P_i(n)]$ be the list of initial assumptions involving n in the box containing $n \neq 0$, then the statement on which we want to apply the induction principle is $P(n)$: $[P_1(n), ..., P_i(n)] \Longrightarrow$ `False`. We add $P(n-1)$ to the box containing $n \neq 0$. Resolving that box in a way that depends on $P(n-1)$ means $P(n-1) \Longrightarrow P(n)$ is proved. From $n \neq 0$

we may obtain $P(0)$. Together they imply $P(n)$ for all values of n using the induction principle. So the dependence on $P(n-1)$ can be removed from the derived contradiction. Strong induction and induction with other variables held arbitrary are also implemented in similar ways (strong induction is triggered by the presence of a proposition $m < n$, where m is a variable created during the proof). In the actual implementation, $P(n-1)$ is transformed into $[P_1(n-1), ..., P_{i-1}(n-1)] \Longrightarrow \neg P_i(n-1)$, for some choice of P_i, before being added.

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

1.7 Examples

We give three examples on how the program can prove selected theorems in `HOL/Number_Theory/Primes.thy`. There are often multiple paths to proving a theorem, and which path the program takes depends on details of the implementation. We only describe one possible path.

1.7.1 Example 1: `prime_odd_nat`

The statement is `prime p` $\Longrightarrow$ `p > 2` $\Longrightarrow$ `odd p`.

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

1. From `TERM (prime p)`, add `prime p = (1 < p ∧ ∀m. m dvd p ⟶ m = 1 ∨ m = p)`.
2. Using the equality from step 1, `prime p` matches pattern `?A ∧ ?B`. From this, we conclude `1 < p` and `∀m. m dvd p ⟶ m = 1 ∨ m = p`.
3. From `∀m. m dvd p ⟶ m = 1 ∨ m = p`, add implication `?m dvd p` $\Longrightarrow$ `?m = 1 ∨ ?m = p`.
4. The implication in step 3 matches the initial assumption `2 dvd p`, creating `2 = 1 ∨ 2 = p`.
5. The disjunction in step 4 triggers a case analysis, creating primitive box 1 under {0} with initial assumption `2 = 1`.
6. In box {1}, `2 = 1` is trivially a contradiction. This resolves box {1}, adding `2 ≠ 1` to box {0}.
7. From `2 ≠ 1` and `2 = 1 ∨ 2 = p`, conclude `2 = p` in box {0}.
8. The equality `2 = p` allows `p > 2` to be matched with pattern `?n > ?n`, which gives a contradiction, resolving box {0}.

In an actually run of AUTO2, other items are added, such as expanding the definition of `2 dvd p`. This can be expected for a search process. The proof finished after 46 steps in less than 1 second.

1.7.2 Example 2: `not_prime_eq_prod_nat`

The statement is `n > 1` $\Longrightarrow$ ¬ `prime n` $\Longrightarrow$ ∃`m k. n = m * k` ∧ `1 < m` ∧ `m < n` ∧ `1 < k` ∧ `k < n`.

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

1. From `TERM (prime n)`, add `prime n = (1 < n` ∧ ∀`m. m dvd n` ⟶ `m = 1` ∨ `m = n)`.
2. ¬ `prime n` matches ¬ `(?A` ∧ `?B)`. Applying de Morgan's law, we get proposition ¬ `1 < n` ∨ ¬ ∀`m. m dvd n` ⟶ `m = 1` ∨ `m = n`.
3. The proposition from step 2 and initial assumption `n > 1` matches [¬`?A` ∨ `?B`, `?A`], to derive ¬ ∀`m. m dvd n` ⟶ `m = 1` ∨ `m = n`.
4. `TERM (`¬ ∀`m. m dvd n` ⟶ `m = 1` ∨ `m = n)` matches `TERM (`¬ ∀`x. ?A)`, adding an equality rewriting it to ∃`m.` ¬ `(m dvd n` ⟶ `m = 1` ∨ `m = n)`.
5. The result of step 4 matches pattern ∃`x. ?A(x)`. Using the existence fact, we get proposition ¬ `(m dvd n` ⟶ `m = 1` ∨ `m = n)`, using `m` as a fresh variable.
6. From step 5, a goal is to prove `m dvd n` ⟶ `m = 1` ∨ `m = n`. So we may add `m dvd n` as a fact and `m = 1` ∨ `m = n` as a goal (that is, add ¬ `(m = 1` ∨ `m = n)` as a fact). Applying de Morgan's law to the latter, add `m` ≠ `1` and `m` ≠ `n` as facts.
7. `m dvd n` (from step 6) rewrites to ∃`k. n = m * k`. Using the existence statement, add proposition `n = m * k`, using `k` as a fresh variable.
8. From `n = m * k`, and `n > 0` (which follows from `n > 1`), we get `1` ≤ `m` ≤ `n` and `1` ≤ `k` ≤ `n`.
9. Since a goal is to show ∃`m k. n = m * k` ∧ `1 < m` ∧ `m < n` ∧ `1 < k` ∧ `k < n`, it suffices to show `1 < m` ∧ `m < n` ∧ `1 < k` ∧ `k < n` (the negation of this proposition is added as a fact).
10. de Morgan's law rewrites ¬ `(1 < m` ∧ `m < n` ∧ `1 < k` ∧ `k < n)` to ¬ `(1 < m)` ∨ ¬ `(m < n` ∧ `1 < k` ∧ `k < n)`. New primitive box 1 is created under {0} with assumption ¬ `1 < m`.
11. In box {1}, since a goal is to show `1 < m`, and `m` ≠ `1` is known in box {0}, it suffices to show `1` ≤ `m` in box {1}. But this is known from step 8. So box {1} is resolved. The propositions `1 < m` and ¬ `(m < n` ∧ `1 < k` ∧ `k < n)` are added to box {0}.

12. Again applying de Morgan's law, we get $\neg$ `m` $<$ `n` $\vee \neg$ `(1` $<$ `k` $\wedge$ `k` $<$ `n)` in box $\{0\}$. New primitive box 2 is created under $\{0\}$ with goal `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 $\neg$ `(1` $<$ `k` $\wedge$ `k` $<$ `n)` to $\{0\}$.

13. Again applying de Morgan's law, we get `(`$\neg$ `1` $<$ `k)` $\vee$ `(`$\neg$ `k` $<$ `n)`. Primitive box 3 is created under $\{0\}$ with goal `1` $<$ `k`.

14. Since `1` $\leq$ `k` is known from step 8, it suffices to prove `1` $\neq$ `k`. Equivalently, we may add `1 = k` as a fact.

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

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

In the actual run, the proof finished in 110 steps, taking approximately 4 seconds.

This example shows how the goal of proving the conjunction of four expressions is transformed via de Morgan's law to checking four cases. One could of course directly add proof steps that handle conjunctive goals in addition to those handling disjunctive facts, but having de Morgan's law is more general.

1.7.3 Example 3: `prime_power_mult`

As a final 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 is `prime p` $\implies$ `x * y = p`^{k} $\implies \exists$`i j. x = p`$^i \wedge$ `y = p`^{j}.

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

2. On seeing goal $\exists$`i. x = p`$^i \wedge \exists$`j. y = p`^{j}, primitive box 1 is created under $\{0\}$ with goal $\exists$`i. x = p`^{i} (via de Morgan's law and case analysis, similar to the second example).

3. The definition of `p`^{k} as `if k = 0 then 1 else p * p`$^{k-1}$ triggers the creation of primitive box 2 under $\{0\}$ with assumption `k = 0`. This acts as the first step of both simple induction and case analysis on `k = 0` and `k` $\neq$ `0`. We omit how box $\{2\}$ is resolved. Once it is, `k` $\neq$ `0` is added to box $\{0\}$.

4. On seeing `k` $\neq$ `0`, invoke induction generalizing other variables, choosing variables `x` and `y` to generalize (we do not generalize `p` here since it appears alone in the initial assumption `prime p`. A more comprehensive analysis

of which inductions to take is one task for the future). This adds proposition $\forall$`x y. x * y = p`$^{k-1} \longrightarrow \exists$`i j. x = p`$^i \wedge$ `y = p`^{j} to box {0}. Using this we add implication `?x * ?y = p`$^{k-1} \Longrightarrow \exists$`i j. ?x = p`$^i \wedge$ `?y = p`^{j}.

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

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

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

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

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

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

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

12. The goal $\exists$`i. x = p`^{i} in box {1} generates the implication `x = p`$^{?i} \Longrightarrow$ `False`. Since the result of step 11 matches the left side of this implication, box {1, 3} is resolved.

13. Resolving box {1, 3} adds facts $\exists$`i. x = p`^{i} to box {3} and $\neg$ `p dvd x` to box {1}. Then goal $\exists$`j. y = p`^{j} is added to box {3}.

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

15. Checking the case `p dvd y` proceeds in box {0} in a way similar to checking the case `p dvd x` in box {3}, again using the case analysis provided by primitive box 1. The rest is omitted.

The pattern of case analysis using primitive boxes 1 and 3 here is essentially as described at the end of the section on the box lattice. In an actual run of AUTO2, the program made many detours, including attempts to resolve the case $k = 1$, and after succeeding, the case $k = 2$. This is not necessarily a bug, as the case $k = 1$ sometimes deserves special consideration (for example, in proving the theorem `prime p`$^n \longleftrightarrow$ `n = 1` $\wedge$ `prime p`). Still, the program finished in 426 steps, taking approximately 16 seconds.

2 Implementation

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

2.1 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 `subterms.ML`. The file 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`.

2.2 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 a pattern with an expression modulo equivalence. The rewrite table is implemented in `rewrite.ML`.

The core data structures in a 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 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.

2.2.1 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, 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.

2.2.2 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.

2.2.3 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 (concrete) 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 `rewrite.ML` for the full details.

2.2.4 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` in `rewrite.ML`.

2.2.5 Matching an equality pattern

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

2.3 More on box items

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

```
datatype raw_item = Init of term * bool
                  | FreeVar of term
                  | Handler of term * retro_handler
                  | Fact of thm
```

The four possible types of raw items are:

- `Init` (t, b) specifies an initial assumption with term t (of type `prop`), which is considered as an assumption t if $b =$ `false`, and a goal $\neg\ t$ if $b =$ `true`. The boolean field is used only when assembling results when a box is resolved, and for finding a good form for the induction hypothesis.

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

- `Handler` $(t, handler)$ declares that the retro-handler *handler* is responsible for removing any dependence on t in the theorem that results when a box is resolved.

- `Fact` th represents one of the three types of items discussed above. The statement (`Thm.prop_of`) of th is `Trueprop` P for a proposition P, `TERM` t for a term t, and $A \Longrightarrow B$ for an implication. Each hypothesis (`Thm.hyps_of`)

of *th* must be either an initial assumption or a term with a registered handler, either in the current box or in one of the ancestor boxes.

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

2.4 Boxes, Updates, and Status

A box is defined as a list of box items. Functions managing this list is defined in structure `Box`.

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

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

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

Here `lat` is a data structure recording the inheritance relations in the box lattice, as well as keeping a list of resolved boxes (implemented in `box_id.ML`). `boxes` is a table mapping box IDs to boxes (lists of box items). `queue` is the priority queue of future updates. `rewrites` is the rewrite table. Finally, `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`.

2.5 Scoring

The scoring function, computing the score of a new update in terms of the scores of dependent items, is contained in structure `Scores`. Currently the scoring function is very simple. In most cases, the score of the new item is the maximum score of the dependent items, plus the size (`Term.size_of_term`) of the proposition. The idea is that we are less willing to add long-winded propositions than short ones. The adjustments upon this are as follows. New variables cost 10. New boxes cost 20. Resolving a box cost -1 (so it is always done first). Implications cost 0 (since the cost is already counted when adding

the $\forall$ or $\exists$ item producing the implication). If adding to a box that is the intersection of $n > 1$ primitive boxes, add cost $5(n-1)$.

In the future, we can make the score depend on the proof step used, reflecting the heuristic that some steps are more attractive than others. Various machine learning techniques can then be used to automatically adjust the parameters computing the score for the best performance.

2.6 Proof steps

In this section we discuss proof steps and the existing facility on writing them. An object of type `proofstep_fn` is the actual function the performs matching with items and producing the new `raw_update` objects. The type is defined as `box_item list` $\rightarrow$ `rewrite_type` $\rightarrow$ `status` $\rightarrow$ `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 the state other than the Isabelle context. They 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 `rewrite_type` $\rightarrow$ `box_item` $\rightarrow$ `bool`. It represents a filter on whether it is possible for an item to serve as the first or second input to a proof step. If an item A cannot possibly be the first input to a proof step, there is no point invoking the proof step on (A, B), for any other B in the state. This is used in function `process_fact_prfsteps` for performance optimizations (`pre_filter` on proof steps taking one item as input is not very useful, since it usually takes the same amount of time for the proof step to reject the item as to run the filter).

The type `proofstep` is defined as a record

{`name: string`, `filt: pre_filter list`, `func: proofstep_fn`}.

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

In addition to defining the types, the structure `ProofStep` also provides some facility to help writing proof steps. While a general proof step can be arbitrarily complicated, most simply consists of applying some theorem when certain conditions are met. These proof steps can be written by specifying the necessary data.

We now look at some examples of writing proof steps.

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

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

```
    Filter (neq_filter "m" "n"),
    Filter (neq_filter "n" "p"),
    Filter (neq_filter "m" "p")]))
```

The function `add_gen_prfstep` accepts a pair (*name*, *descs*), where *name* is the name of the proof step, and *descs* is a list of *descriptors* specifying the proof step. It directly calls the function `gen_prfstep` to generate the proof step, and adds it to the Isabelle theory. Here the name of the proof step is `dvd_transitive`. The first two descriptors specify the patterns to be matched: `(?m::nat) dvd ?n` and `(?n::nat) dvd ?p`. The third descriptor specify the update produced: adding a new proposition `(?m::nat) dvd ?p`. This is justified by the theorem with name `dvd_transitive`. The statement of this theorem is:

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

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

where the theorem `le_contra` 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.

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

```
val _ = Theory.setup (
  add_rewrite_rule (
    [WithTerm @{term_pat "(?a::nat) ^ 1"},
```

```
     Filter (neqt_filter "a" @{term "1::nat"})],
    @{thm power_one_right}))
```

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

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

We require the term pattern to agree with one side of the rewritting 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.

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

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

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

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

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

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

The theorem `exists_intro2` is:

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

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

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

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

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

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

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

which is the one needed for the proof.

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

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

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

`¬ (∃x. P x ∧ Q x) ⟹ P x ⟹ ¬ (Q x)`

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

9. (Generating theorem on demand). The proof step `compare_consts` is written as follows:

```
val _ = Theory.setup (
  add_prfstep_thm_fn (
    "compare_consts",
    [WithFact @{term_pat "(?NUMC1::nat) = ?NUMC2"},
     Filter (fn _ => fn (_, inst) =>
       HOLogic.dest_number (lookup_instn inst ("NUMC", 1)) <>
       HOLogic.dest_number (lookup_instn inst ("NUMC", 2)))],
    (fn ctxt => fn (_, ths) =>
        let
          val goal = Logic.mk_implies (
              Thm.prop_of (the_single ths), @{prop "False"})
        in
          ths MRS (Goal.prove ctxt [] [] goal
                     (K (Arith_Data.arith_tac ctxt 1)))
        end)))
```

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 Isabelle tactic `Arith_Data.arith_tac` to produce a theorem deriving the contradiction (the theorem returned by the body is `False`, which `add_prfstep_thm_fn` will use the produce an update resolving the box. If the theorem returned by the body is not `False`, an update adding it as a new item will be produced).

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