A human-oriented term rewriting system

Edward W. Ayers[1], W.T. Gowers[1], and Mateja Jamnik[2]

[1] DPMMS, University of Cambridge
[2] Department of Computer Science and Technology, University of Cambridge

Abstract. We introduce a fully automatic system, implemented in the Lean theorem prover, that solves equality problems of everyday mathematics. Our overriding priority in devising the system is that it should construct proofs of equality in a way that is similar to that of humans. A second goal is that the methods it uses should be domain independent. The basic strategy of the system is to operate with a subtask stack: whenever there is no clear way of making progress towards the task at the top of the stack, the program finds a promising subtask, such as rewriting a subterm, and places that at the top of the stack instead. Heuristics guide the choice of promising subtasks and the rewriting process. This makes proofs more human-like by breaking the problem into tasks in the way that a human would. We show that our system can prove equality theorems simply, without having to preselect or orient rewrite rules as in standard theorem provers, and without having to invoke heavy duty tools for performing simple reasoning.

1 Introduction

In mathematical proofs one often finds chains of expressions linked by equalities. They are designed to show that the first expression in the chain is equal to the last one, with all the equalities being sufficiently obvious to the reader that no further justification is needed. For example, suppose that one wishes to prove that given a linear map A, its adjoint $A^\dagger$ is linear. To do so one typically provides the following equality chain for all vectors x and all dual vectors u, v:

$$\begin{aligned}\langle A^\dagger(u+v),x\rangle = \langle u+v,Ax\rangle = \langle u,Ax\rangle + \langle v,Ax\rangle = \\ \langle A^\dagger u,x\rangle + \langle v,Ax\rangle = \langle A^\dagger u,x\rangle + \langle A^\dagger v,x\rangle = \langle A^\dagger u + A^\dagger v,x\rangle \quad (1)\end{aligned}$$

Here, $\langle\cdot,\cdot\rangle$ is the inner product taking a dual vector and a vector to a real number. The equations that we can compose our reasoning chain from (e.g., $\langle A^\dagger a,b\rangle = \langle a,Ab\rangle$) are called *rewrite rules.*

A central part of automated theorem proving (ATP) is constructing such equality proofs automatically. This can be done with well-researched techniques from the field of *term rewriting systems* [1]. These techniques take advantage of the fact that computers can perform many operations per second, and large search spaces can be explored quickly, though heuristic functions are still needed to prevent a combinatorial explosion. Many domains – such as checking that two expressions are equal using the ring axioms – also have specialised decision procedures available for them. We will call these approaches to solving equalities *machine-oriented*.

We wish to investigate alternative ways of producing equality proofs. We do not wish to compete with machine-oriented techniques to prove more theorems or prove them faster. Instead, we are motivated by a desire to prove theorems in a different way which better captures the abstract reasoning that seems to occur in the mind of a human.

Why bother finding proofs that are more human-oriented? One answer is purely about efficiency: the runtimes of existing ATP methods do not scale well with the number of competing rules introduced, as one would expect of algorithms that make use of significant amounts of brute-force search. If we can devise new architectures that solve simple equalities with less search, then it may be possible to scale up these techniques to larger problems and improve the efficiency of established ATP methods.

Another reason is that it makes interactive theorem proving easier for the non-specialist. While there is a growing community of mathematicians using formal verification techniques, ATP is met with indifference by the majority of mathematicians [6]. A large part of the reason for this is that the user who wishes to prove a lemma in a proof assistant often has to explicitly provide proofs of intermediate results that would be omitted in a mathematical document. Tools such as Isabelle's Sledgehammer [3] have ameliorated the problem to a large extent, but finding proofs can be slow and the resulting tactics that Sledgehammer recommends are still somewhat cryptic and add clutter. By developing automation that can solve problems that mathematicians find easy, we can contribute to the goal of producing verified proofs that are as easy to read and write as informal ones.

With this in mind, our goals are to create an algorithm which:

- can solve simple equality problems of the kind that an undergraduate might find easy;
- does not encode any domain-specific knowledge of mathematics, that is, it does not invoke specialised procedures if it detects that the problem lies in a particular domain such as Presburger arithmetic;
- is efficient in the sense that it does not store a large state and does not perform a significant search when a human would not.

In this paper we present the `subtask` algorithm which has some success with respect to the above goals. The algorithm is written in Lean 3 [12] and can be found at https://github.com/EdAyers/lean-subtask. In the remainder of the paper we give a motivating example in § 2 followed by a description of the algorithm in § 3. The algorithm is then contrasted with existing approaches in § 4 and evaluated against the above goals in § 5. Conclusions and further work are contemplated in § 6.

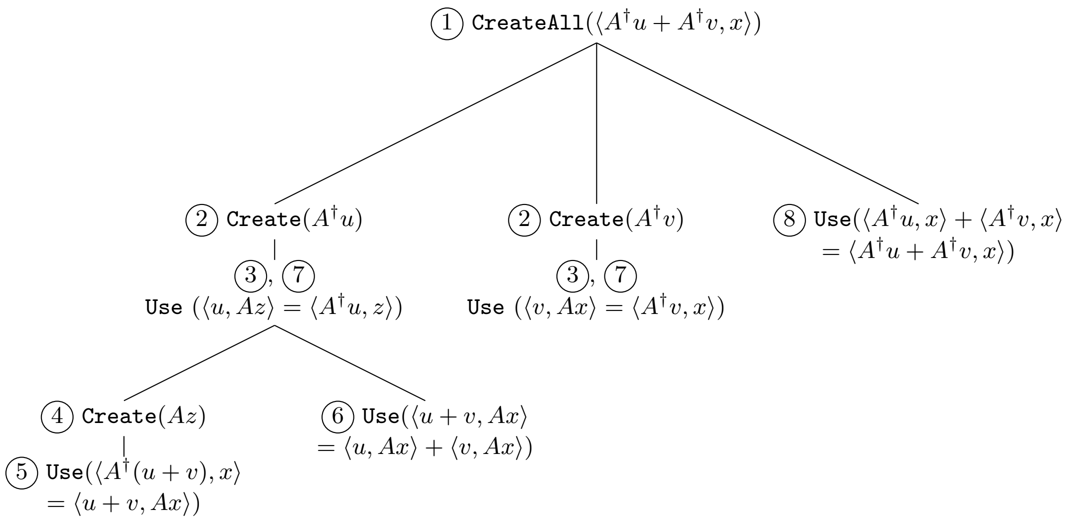
2 Example

Let us begin with the example in elementary linear algebra mentioned above. We have to prove the equality $\langle A^\dagger(u+v), x\rangle = \langle A^\dagger u + A^\dagger v, x\rangle$.

To do this, a human's (not fully conscious) thought process might proceed as follows.

1. I need to create the expression $\langle A^\dagger u + A^\dagger v, x\rangle$.
2. In particular, I need to make the subexpressions $A^\dagger u$ and $A^\dagger v$.
3. The only sensible way I can get these is to use the definition $\langle w, Az\rangle = \langle A^\dagger w, z\rangle$ applied with $w = u$ and v, and presumably with $z = x$.
4. In particular, I'll need to make the subterm Az for some z.
5. I can do that straight away: $\langle A^\dagger(u+v), x\rangle = \langle u+v, Ax\rangle$.
6. Now I'm in a position to obtain the subexpressions $\langle u, Ax\rangle$ and $\langle v, Ax\rangle$ I wanted in step 3, so let me do that using bilinearity: $\langle u+v, Ax\rangle = \langle u, Ax\rangle + \langle v, Ax\rangle$.
7. And now I can get the subexpressions $A^\dagger u$ and $A^\dagger v$ I wanted even earlier in step 2, so let me do that: $\langle u, Ax\rangle + \langle v, Ax\rangle = \langle A^\dagger u, x\rangle + \langle A^\dagger v, x\rangle$.
8. And with one application of bilinearity I'm home: $\langle A^\dagger u, x\rangle + \langle A^\dagger v, x\rangle = \langle A^\dagger u + A^\dagger v, x\rangle$.

The key aspect of the above kind of thought process that we wish to model is the setting of intermediate aims, such as obtaining certain subexpressions when we do not immediately see how to obtain the entire expression. We do this by creating a tree of subtasks.



① CreateAll($\langle A^\dagger u + A^\dagger v, x\rangle$)
② Create($A^\dagger u$)
② Create($A^\dagger v$)
⑧ Use($\langle A^\dagger u, x\rangle + \langle A^\dagger v, x\rangle = \langle A^\dagger u + A^\dagger v, x\rangle$)
③, ⑦ Use ($\langle u, Az\rangle = \langle A^\dagger u, z\rangle$)
③, ⑦ Use ($\langle v, Ax\rangle = \langle A^\dagger v, x\rangle$)
④ Create(Az)
⑥ Use($\langle u+v, Ax\rangle = \langle u, Ax\rangle + \langle v, Ax\rangle$)
⑤ Use($\langle A^\dagger(u+v), x\rangle = \langle u+v, Ax\rangle$)

Fig. 1: The subtask tree for solving $\langle A^\dagger(u+v), x\rangle = \langle A^\dagger u + A^\dagger v, x\rangle$. Circled numbers correspond to steps in the above list.

The tree in Figure 1 represents what the algorithm does with the additivity-of-adjoint problem. It starts with the subtask `CreateAll`$(\langle A^\dagger u + A^\dagger v, x\rangle)$. Since it cannot achieve that in one go, it creates some subtasks and then chooses the one that is most promising: later in § 3.1 we shall give details about how it generates and evaluates possible choices. In this case the most promising subtask is `Create`$(A^\dagger u)$, so it selects that and identifies a rewrite rule – the basic definition of adjoint – that can achieve it. The z that appears is a metavariable that will in due course be set to x. (It will typically also find a number of 'silly' possibilities, not depicted here, which are dismissed by the scoring system.) When it has done that, it has a new subtask which is to create the left-hand side of the rule. It cannot do that in one go, so it creates new subtasks and so on. The process outlined in this example is the one which we want our algorithm to reflect.

3 Design of the algorithm

The `subtask` algorithm acts on a tree of *tasks* (as depicted in Figure 1) and an expression called the *current expression* (CE). A task is any object which implements the following methods:

- `refine : task -> list task`
- `test : task -> bool` which returns true when the task is *achieved* for the current expression.
- optionally, `execute : task -> unit` which updates the current expression `x` to `y` by providing a proof of `x = y`. Tasks with `execute` methods are called *strategies*. In this case, `test` returns true when `execute` can be applied successfully.

The main tasks are given in Table 1, however more are added to the software. For example `ReduceDistance`(x, y) will greedily apply any rewrite that causes x and y to be closer in the parse tree. The algorithm is summarised in the pseudocode in Figure 2.

In the `explore` phase, we take a task X on the tree and `refine` it to produce a list of child tasks $C_1, C_2 \cdots$. We add these to the task tree if they are not already present on it. We then score the strategies $S_1, S_2 \cdots$ in this list – that is, score the children where `execute` is defined. The score is intended to represent the likelihood of the strategy being successful and is determined by heuristics discussed in § 3.1. The reason why the algorithm focusses on strategies before non-strategies is a heuristic that seems to work well in practice. The underlying idea behind the heuristic is that often the first sensible strategy found is enough of a signpost to solve simple problems. That is, once one has found one plausible strategy of solving a simple problem it is often fruitful to stop looking for other strategies which achieve the same thing and to get on with finding a way of performing the new strategy.

If the overall score is above zero then add a backtrack point and take the highest-scoring strategy S.

	`refine`	`test`	`execute`
`CreateAll(e)`	Returns a list of `Create(b)` subtasks where each `b` is a minimal subterm of `e` not present in the CE.	True whenever the CE is `e`.	none
`Create(e)`	Returns a list `Use(a = b)` subtasks where each `e` occurs within `a`.	True whenever the CE contains `e`.	none
`Use(a = b)`	Returns a list of `Create(e)` subtasks where each `e` is a minimal subterm of `a` not present in the CE.	True whenever the rule `a = b` can be applied to the CE.	Apply `a = b` to the CE.

Table 1: Main tasks

```
function explore(X : task) {
  children <- refine(X)
  foreach (C in children) {
    if (C is not on the task tree) {
      add C as a child node of X
    }
  }
  strategies <- children.strategies
  overall_score <- score(strategies)
  if (overall_score > 0) {
    add a backtrack point
    S <- strategies.highest_scoring
    ascend(S)
  } else {
    explore a non-strategy
    child of X or else backtrack
  }
}
```

```
function ascend(X : task) {
  if (X is a strategy) {
    if (test(X)) {
      execute(X)
      ascend(parent(X))
    } else {
      explore(X)
    }
  } else {
    if (test(X)) {
      if (X is the root task) {
        success
      } else {
        ascend(parent(X))
      }
    } else {
      explore(X)
    }
  }
}
```

Fig. 2: Pseudocode for the subtask algorithm.

If `test`(S) is false then `explore` S otherwise execute S and ascend S's parents until a task Y is found that can not be achieved then `explore` Y. Otherwise if the overall score is less than or equal to zero then `explore` a non-strategy child task of X or backtrack if none exist. The backtracking procedure works by taking the list of backtracking points and choosing the one with the highest overall score if the failed branch is removed.

To find `l = r`, the algorithm is initialised with the tree `CreateAll(r)` and the current expression `l`. We then run `execute(CreateAll(r))` until a timeout is reached or we run out of backtracking points.

3.1 Heuristics

Both lists of strategies and individual strategies are scored using a heuristic to guide the exploration of the tree. The system prioritises strategies if they:

- achieve a task higher in the task tree;
- achieve a task on a different branch of the task tree;
- have a high degree of term overlap with the current expression (this is measured using symbol counting and finding largest common subterms);
- use local hypotheses;
- can be achieved in one rewrite step from the current expression.

The overall score heuristic evaluates sets of strategies. If there is only one strategy then it scores 10. If there are multiple strategies, it discards any scoring less than -5. If there are positive-scoring strategies then all negative-scoring strategies are discarded. The overall score is then set to be 5 minus the number of strategies in the list. The intention of this simple procedure is that we should prefer smaller sets of strategies, even if their scores are bad because it limits choice in what to do next.

4 Related work

4.1 Term rewriting

One way to find equality proofs is to perform a graph search using a heuristic. This is the approach of the `rewrite-search` algorithm [8], which uses the heuristic of string edit-distance between the strings two pretty-printed expressions. The `rewrite-search` algorithm does capture some human-like properties in the heuristic, since the pretty printed expressions are intended for human consumption. Our algorithm is different from `rewrite-search` in that we guide search according to achieving sequences of tasks. Since both our software and `rewrite-search` are written in Lean, some future work could be to investigate a combination of both systems.

A term rewriting system (TRS) R is a set of oriented rewrite rules. There are many techniques available for turning a set of rewrite rules in to procedures that check whether two terms are equal. One technique is *completion*, where R is converted into an equivalent TRS R' that is *convergent*. This means that any two expressions a, b are equal under R if and only if repeated application of rules in R' to a and b will produce the same expression. Finding equivalent convergent systems, if not by hand, is usually done by finding decreasing orderings on terms and using Knuth-Bendix completion. When such a system exists, automated rewriting systems can use these techniques to quickly find proofs, but the proofs are often overly long and needlessly expand terms.

Another method is rewrite tables, where a lookup table of representatives for terms is stored in a way that allows for two terms to be matched through a series of lookups.

Both completion and rewrite tables can be considered machine-oriented because they rely on large datastructures and systematic applications of rewrite rules. Such methods are certainly highly useful, but they can hardly be said to capture the process by which humans reason.

Finally, there are many normalisation and decision procedures for particular domains, for example on rings [7]. Domain specific procedures do not satisfy our criterion of generality.

4.2 Proof planning

As mentioned earlier, our approach is similar to that of proof planning [4]. AI planning in its most general conception [9] is the process of searching a graph G using plan-space rather than by searching it directly. In a typical planning system, each point in plan-space is a DAG of objects called *ground operators* or *methods*, each of which has a mapping to paths in G. Each ground operator is equipped with predicates on the vertices of G called *pre/post-conditions*. Various AI planning methods such as GRAPHPLAN [2] can be employed to discover a partial ordering of these methods, which can then be used to construct a path in G. This procedure applied to the problem of finding proofs is known as proof planning. The main issue with proof planning [5] is that it is difficult to identify sets of conditions and methods that do not cause the plan space to be too large or disconnected. However, in this paper we are not trying to construct plans for entire proofs, but just to model the thought processes of humans when solving simple equalities.

Proof planning in the domain of finding equalities frequently involves a technique called *rippling*, in which an expression is annotated with additional structure determined by the differences between the two sides of the equation that directs the rewriting process. In our system we avoid using rippling because of our concern for generality: for finding chains of equalities, subtasks achieve similar results and are less tied to particular domains.

Our approach also shares properties with Hierarchical Task Networks (HTN) [11,13] used to drive the behaviour of artificial agents such as the ICARUS architecture [10]. Starting tasks are broken down into subtasks, which are then used to find fine-grained methods for achieving the original tasks.

The main difference between our approach and proof planning and hierarchical task networks is that our algorithm is greedier: we generate enough of a plan to have little doubt what the first rewrite rule in the sequence should be, and no more. We believe that this reflects how humans reason for solving simple problems: favouring just enough planning to decide on a good first step, and then planning further only once the step is completed and new information is revealed.

5 Evaluation

Our ultimate motivation is to make an algorithm that behaves as a human mathematician would. We do not wish to claim that we have fully achieved this, but we can evaluate our algorithm with respect to some general goals that we mentioned in § 1.

– Scope: can it solve simple equations?
– Generality: does it avoid techniques specific to a particular area of mathematics?
– Reduced search space: does the algorithm avoid search when finding proofs that humans can find easily without search?
– Straightforwardness of proofs: for easy problems, does it give a proof that an experienced human mathematician might give?

Our method of evaluation is to use the algorithm implemented as a tactic in Lean on a library of thirty or so example problems. This is not large enough for a substantial quantitative comparison with existing methods, but we can still investigate some properties of the algorithm. The source code also contains many examples which are outside the abilities of the current implementation of the algorithm. Some ways to address these issues are discussed in § 6.

Table 2 gives some selected examples. These are all problems that the algorithm can solve with no backtracking.

From this table we can see that the algorithm solves problems from several different domains. We did not encode any decision procedures for monoids or rings. In fact we did not even include reasoning under associativity and commutativity, although we are not in principle against extending the algorithm to do this. The input to the algorithm is simply a list of over 100 axioms and equations for sets, rings, groups and vector spaces which can be found in the file `equate.lean` in the source code. Thus, the algorithm exhibits considerable generality.

All of the solutions to the above examples are found without backtracking, which adds support to the claim that our algorithm requires less search. There are other examples in the source where backtracking occurs, so there is still some work to be done on choosing scoring heuristics here.

Our final criterion is that the proofs are more straightforward than those produced by machine-oriented special purpose tactics. This is a somewhat subjective measure but there are some proxies that indicate that `subtasks` can be used to generate simpler proofs.

To illustrate this point, consider the problem of proving $(x+y)^2+(x+z)^2 = (z+x)^2+(y+x)^2$ within ring theory. We choose this example because it is easy for a human to spot how to do it with three uses of commutativity, but it is easy

problem	**#steps**	**location**
$l\ s$: `list` $\texttt{rev}(l \mathbin{++} s) = \texttt{rev}(s) \mathbin{++} \texttt{rev}(l)$ $\texttt{rev}(a :: l) = \texttt{rev}(l) \mathbin{++} [a]$ $\vdash \texttt{rev}(h :: l \mathbin{++} s) = \texttt{rev}(s) \mathbin{++} \texttt{rev}(h :: l)$	5	`datatypes.lean/rev_app_rev`
a : monoid element $a^{(m+n)} = a^m * a^n$ $\vdash a^{\texttt{succ}(m)+n} = a^{\texttt{succ}(m)} * a^n$	8	`groups.lean/my_pow_add`
$a\ b$: ring element $a * d = c * b$ $c * f = e * d$ $\vdash d * (a * f) = d * (e * b)$	9	`rat.lean`
$a\ b$: ring element $\vdash (a + b) * (a + b) = a * a + 2 * (a * b) + b * b$	7	`rings.lean/sumsq_with_equate`
$A\ B\ C\ X$: set $\vdash X \setminus (B \cup C) = (X \setminus B) \setminus C$	4	`sets.lean/example_4`

Table 2: `subtask`'s performance on some example problems. "# steps" gives the number of rewrite steps in the final proof. "location" gives the file and declaration name of the example in the source code.

for a program to be led astray by expanding the squares. `subtask` proves this equality with 3 uses of commutativity and with no backtracking or expansion of the squares. This is an example where domain specific tactics do worse than `subtask`, the `ring` tactic for reasoning on problems in commutative rings will produce a proof by expanding out the squares. The built-in tactics `ac_refl` and `blast` in Lean which reason under associativity and commutativity both use commutativity 5 times. If one is simply interested in verification, then such a result is perfectly acceptable. However, we are primarily interested in modelling how humans would solve such an equality, so we want our algorithm not to perform unnecessary steps such as this.

It is difficult to fairly compare the speed of `subtask` in the current implementation because it is compiled to Lean bytecode which is much slower than native built-in tactics that are written in C++. However it is worth noting that, even with this handicap, `subtask` takes 1900ms to find the above proof whereas `ac_refl` and `blast` take 600ms and 900ms respectively.

There are still proofs generated by `subtask` that are not straightforward. For example, the lemma $(xz)(z^{-1}y) = xy$ in group theory is proved by `subtask` with a superfluous use of the rule $e = xx^{-1}$. We hope that some of these defects will be ironed out in future versions of the program.

6 Conclusions and Further Work

In this paper, we introduced a new, task-based approach to finding equalities in proofs and provided a demonstration of the approach by building the `subtask` tactic in Lean. We show that the approach can solve simple equality proofs with very little search. Our hope is that our work will renew interest in proof planning and spark interest in human-oriented reasoning for at least some classes of problems.

In future work, we wish to add more subtasks and better heuristics for scoring them. The framework we outlined here allows for easy experimentation with such different sets of heuristics and subtasks. In this way, we also wish to make the subtask framework extensible by users, so that they may add their own custom subtasks and scoring functions.

There are times when the algorithm fails and needs guidance from the user. We wish to study further how the subtask paradigm might be used to enable more human-friendly interactivity than is currently possible. For example, in real mathematical textbooks, if an equality step is not obvious, a relevant lemma will be mentioned. Similarly, we wish to investigate ways of passing 'hint' subtasks to the tactic. For example, when proving $x*y = (x*z)*(z^{-1}*y)$, the algorithm will typically get stuck (although it can solve the flipped problem), because there are too many ways of creating z. However, the user – upon seeing `subtask` get stuck – could steer the algorithm with a suggested subtask such as `Create`$(x*(z*z^{-1}))$.

Using subtasks should help to give better explanations to the user. The idea of our algorithm is that the first set of strategies in the tree roughly corresponds to the high-level actions that a human would first consider when trying to solve the problem. Thus, the algorithm could use the subtask hierarchy to determine when no further explanation is needed and thereby generate abbreviated proofs of a kind that might be found in mathematical textbooks.

Another potential area to explore is to perform an evaluation survey where students are asked to determine whether an equality proof was generated by our software or a machine.

References

1. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge university press (1999)
2. Blum, A.L., Furst, M.L.: Fast planning through planning graph analysis. Artificial intelligence 90(1-2), 281–300 (1997)
3. Böhme, S., Nipkow, T.: Sledgehammer: judgement day. In: International Joint Conference on Automated Reasoning. pp. 107–121. Springer (2010)
4. Bundy, A.: The use of explicit plans to guide inductive proofs. In: International conference on automated deduction. pp. 111–120. Springer (1988)
5. Bundy, A.: A critique of proof planning. In: Computational Logic: Logic Programming and Beyond, pp. 160–177. Springer (2002)
6. Bundy, A.: Automated theorem provers: a practical tool for the working mathematician? Annals of Mathematics and Artificial Intelligence 61(1), 3 (2011), `http://dx.doi.org/10.1007/s10472-011-9248-8`

7. Grégoire, B., Mahboubi, A.: Proving equalities in a commutative ring done right in coq. In: International Conference on Theorem Proving in Higher Order Logics. pp. 98–113. Springer (2005)
8. Hoek, K., Morrison, S.: lean-rewrite-search repository (2019), `https://github.com/semorrison/lean-rewrite-search`
9. Kambhampati, S., Knoblock, C.A., Yang, Q.: Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning. Artificial Intelligence 76(1), 167–238 (1995)
10. Langley, P., Choi, D., Trivedi, N.: Icarus user's manual. Institute for the Study of Learning and Expertise 2164 (2011)
11. Melis, E., Siekmann, J.: Knowledge-based proof planning. Artificial Intelligence 115(1), 65–105 (1999)
12. de Moura, L., Kong, S., Avigad, J., Van Doorn, F., von Raumer, J.: The lean theorem prover (system description). In: International Conference on Automated Deduction. pp. 378–388. Springer (2015)
13. Tate, A.: Generating project networks. In: Proceedings of the 5th international joint conference on Artificial intelligence-Volume 2. pp. 888–893. Morgan Kaufmann Publishers Inc. (1977)