Twitch: Learning Abstractions for Equational Theorem Proving

Citation for the original published paper (version of record):
Axelrod, G., Johansson, M., Smallbone, N. (2026). Twitch: Learning Abstractions for Equational Theorem Proving. Lecture Notes in Computer Science, 16688 LNCS: 62-79.
http://dx.doi.org/10.1007/978-3-032-32589-1_4

Twitch: Learning Abstractions for Equational Theorem Proving

Guy Axelrod[1,2], Moa Johansson[1,2], and Nicholas Smallbone[1,2(✉)]

[1] Chalmers University of Technology, Göteborg, Sweden
{guya,jomoa,nicsma}@chalmers.se
[2] University of Gothenburg, Göteborg, Sweden

Abstract. Automated theorem provers often perform better when told what shapes of terms are interesting. In this paper we discover interesting term shapes automatically, in the form of *abstractions*, term patterns that occur over and over again in proofs. Our tool *Twitch* produces abstractions automatically and can do so in two ways: (1) from a partial, failed proof of a conjecture; (2) from successful proofs of other theorems in the same domain. Twitch is built on top of Stitch, a tool designed for discovering reusable library functions in program synthesis tasks. We have also extended Twee, an equational theorem prover, to use the generated abstractions. We evaluate Twitch on a set of unit equality (UEQ) problems from TPTP, and show that it proves problems previously unsolved by Twee, as well as yielding speed-ups on many other problems.

Keywords: automated theorem proving · equational theorem proving · completion · abstraction learning · proof-guided search

1 Introduction

Automated theorem provers must navigate a massive search space, choosing the useful inferences and discarding the useless ones. Modern provers therefore rely on sophisticated guidance mechanisms, including finely-tuned clause-selection heuristics [29], strategy portfolios [6], and learning-based guidance [11,17].

A complementary approach, popularized by Otter [22], is to inject human guidance about which *shapes* of terms or clauses are interesting. The user specifies a set of interesting shapes, and generated clauses that match these shapes are preferred over ones that do not. This idea originated in the *weighting strategy* [24] and was later refined by the *resonance strategy* [7,14,37] and *hint strategy* [33,34]. We are attempting to automate this by discovering interesting shapes automatically. We focus on equational reasoning, in particular, the equational prover Twee [30]. Compared with the highly tuned guidance mechanisms available in other ATPs, Twee's heuristics are rather simplistic, relying mostly on the size of the critical pair. This makes Twee a good setting in which to study whether automatically discovered term patterns can provide useful guidance.

A. Biere et al. (Eds.): IJCAR 2026, LNAI 16688, pp. 62–79, 2026.
https://doi.org/10.1007/978-3-032-32589-1_4

To find interesting term shapes *automatically*, we analyse failed proof attempts and proofs of related conjectures to find term shapes which occur often, which we call *abstractions*. We have added support for abstractions to Twee, and with the help of automatically generated abstractions we are able to prove 12 rating-1[1] problems from the TPTP [32], as well as several other problems that Twee previously could not solve (see Sect. 5). Our contributions are:

- A method for automatically generating abstractions from an *unsuccessful attempt* to prove a conjecture (see Sect. 3.1). The idea is to pick out interesting lemmas from the proof attempt, and find patterns of terms that occur repeatedly in the proofs of those lemmas. To find the patterns we use the tool Stitch [10], which attempts to *compress* a set of terms (here from proofs) by introducing auxiliary function definitions as abbreviations.
- A method for generating abstractions from a corpus of successful proofs in the same domain, again using Stitch (see Sect. 3.2). The hypothesis is that patterns of terms that occur repeatedly in the proofs are often good abstractions for other conjectures in the same domain. The aim here is to create a methodology inspired by curriculum learning [8] for proving difficult equational problems: first prove some simpler problems in the same domain, abstract out common patterns appearing in those proofs, and use those abstractions to guide the proof of the difficult problem.
- An implementation of abstractions in the equational theorem prover Twee [30], inspired by the weighting strategy [24] (see Sect. 4).

The code for Twitch as well as all data and experimental results are supplied in the associated repository [1].

1.1 Motivating Example

Let us consider the problem `LAT075-1` from the TPTP, a problem about modular ortholattices in a language with a single function symbol f representing the Sheffer stroke (or NAND operator). The details of this problem are not important here, but what is relevant is that the same kind of term appears repeatedly in

[1] From the TPTP Technical Manual [31, Problem Presentation], the rating "gives the difficulty of the problem, measured relative to state-of-the-art ATP systems [...] The rating is a real number in the range 0.0 to 1.0, where 0.0 means that all state-of-the-art ATP systems can solve the problem [...] and 1.0 means no state-of-the-art ATP system can solve the problem".

the proof that Twee finds. Here is a short fragment of the proof:

$$\begin{aligned}
\ldots &= f(x, f(y, \boxed{f(f(z, \boxed{f(x,x)}), f(z, \boxed{f(x,x)}))}))\\
&= f(x, f(y, f(f(z, \boxed{f(x,x)}), f(\boxed{f(x,x)}, z))))\\
&= f(x, f(y, \boxed{f(f(\boxed{f(x,x)}, z), f(\boxed{f(x,x)}, z))}))\\
&= f(\boxed{f(\boxed{f(x,x)}, \boxed{f(x,x)})}, f(y, \boxed{f(f(\boxed{f(x,x)}, z), f(\boxed{f(x,x)}, z))}))
\end{aligned}$$

As we can see, terms of the form $f(t,t)$, which we have marked with a box above, appear over and over in the proof.[2] Our hypothesis is that *the same term shape* is likely to be common in proofs of other theorems about the Sheffer stroke.

This suggests the following basic approach to proving a difficult theorem. First we analyse a proof of a related theorem to find commonly occurring term patterns. Then, hoping that these term patterns might also occur in the proof of the difficult theorem, we ask the theorem prover to prefer inferences involving those term patterns. The remaining questions are, how can we find these commonly-occurring term patterns, and how can the prover exploit them?

To formalize the notion of *common patterns* in a proof, we imagine *compressing* the proof by introducing new definitions to abbreviate certain terms. For example, if we were to define $g(x) := f(x,x)$, then the proof above could be written more compactly; e.g. the first term could be written as $f(x, f(y, g(f(z, g(x)))))$. Then we ask: what is a *maximally compressive* set of abbreviations, that is, a set of definitions that allows us to express the proof terms as compactly as possible?

The tool Stitch [10] (see Sect. 2) is designed to answer exactly this question. Given a set of terms, it searches for the maximally compressive function definitions. Henceforth, we will call such function definitions *abstractions*. In this case, Stitch would find that $g(x) := f(x,x)$ is a maximally compressive abstraction (and when given the full proof, it also finds other useful abstractions, such as the AND operator $h(x,y) := f(f(x,y), f(x,y))$).

Once we have these abstractions, how can we use them to influence proof search? The simplest option is to add the function definition as an axiom, in this case $\forall x.\, g(x) = f(x,x)$. This allows the prover to rewrite terms of the form $f(t,t)$ to $g(t)$, making the term smaller, so that critical pairs containing the term are more likely to be selected. For `LAT075-1` this approach works well, and reduces the runtime of Twee from $\sim 250s$ to $\sim 10s$.

Adding definitional axioms that abbreviate useful term patterns is a known and useful technique.[3] However, adding new axioms also allows new inferences between them, thus expanding the search space. As we show in Sect. 5.3, adding more than a few such axioms harms the prover's performance.

[2] This is perhaps not so surprising as $f(t,t)$ represents the negation of t.

[3] For example, see the Robbins problem variant `ROB033-1` in the TPTP. Definitional axioms are also used in Twee's goal direction heuristic [30].

Instead, we have extended Twee to natively support abstractions (see Sect. 4). The abstractions influence the heuristic function of the prover, so that inferences matching them are more likely to be selected, but they do not alter the search space itself. Because of this, we can provide many more abstractions, but the individual abstractions may have a smaller effect. In this case, passing $f(x, x)$ as an abstraction reduces the runtime of Twee from $\sim 250s$ to $\sim 130s$.

2 Background: Stitch

Stitch [10] is a system originally designed for library learning in program synthesis – automatically discovering reusable function abstractions from an existing collection of programs. Given a set of terms as input, it constructs functions which generalize parts of the terms, and uses those functions to abbreviate the terms. Stitch attempts to construct *maximally compressive* abstractions; specifically, it maximizes the product of the size of the function and the number of places where the function can be used.

Although Stitch was designed to work on program code, it accepts as input any set of terms. Hence, given a Twee proof, we can give Stitch the set of all terms occurring in the proof and it will find abstractions for it.[4]

Stitch's input language is a higher-order λ-calculus. Thus, we need to translate proof terms into λ-terms, which we do by treating function symbols as constants, and λ-abstracting all free variables. For example, the term $f(x,\ f(y,\ f(f(z, f(x, x)),\ f(z, f(x, x)))))$ from our example is translated into $\lambda x.\lambda y.\lambda z.\ (f\ x\ (f\ y\ (f\ (f\ z\ (f\ x\ x))\ (f\ z\ (f\ x\ x))))))$. In this case, Stitch will return the abstraction $g(\alpha) := (f\ \alpha\ \alpha)$, where α is a *schematic parameter* standing for an arbitrary subterm, which can be used to compress the input term to $\lambda x.\lambda y.\lambda z.\ (f\ x\ (f\ y\ g(f\ z\ g(x))))$. Because the body of g has a first-order shape, it can be translated back directly, yielding the abstraction $f(x, x)$.

In general, Stitch abstractions are arbitrary λ-terms and can be higher-order. In practice, Stitch rarely generates higher-order abstractions in our setting since the input terms are first-order. If it does, we discard the abstraction.

3 Twee + Stitch = Twitch

In this section we present the implementation of Twitch: a system for automating the discovery of reusable abstractions that can be used to help solve hard problems in a given domain. Twitch has two modes of usage: abstractions can be discovered from partial proof attempts as described in Sect. 3.1, or from easier proofs in the same domain, as described in Sect. 3.2. The abstractions found by Twitch are used to adjust the search heuristics in the Twee prover, as described in Sect. 4.

Henceforth, we denote a given input problem – which consists of a set of equations (axioms) and a goal/conjecture equation to prove – by P.

[4] The terms given to Stitch are the top-level terms from Twee proof steps, but Stitch's compression procedure searches for repeated structure inside these terms.

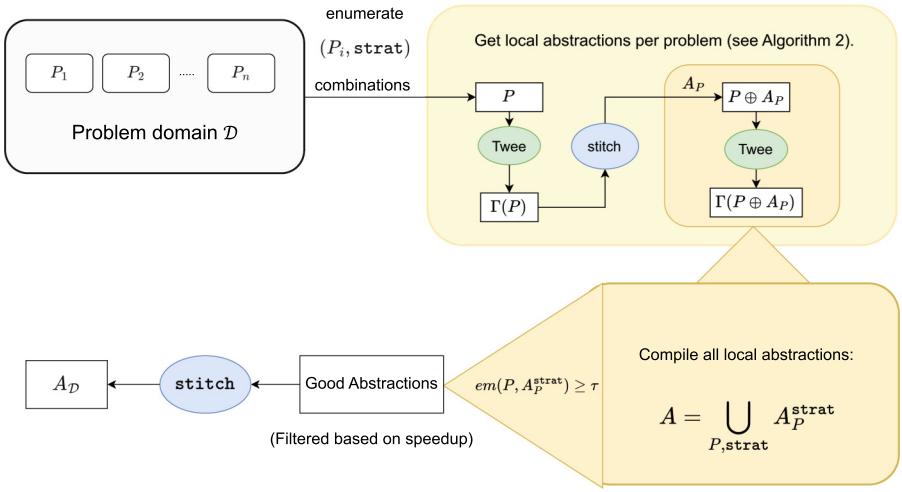


Fig. 1. Overview of domain abstraction generation via Twitch (Algorithm 1 and its subroutine Algorithm 2).

3.1 Abstractions from Partial Proofs

The first approach we explore discovers abstractions given as input a single hard problem P_h, and can be used when we do not have examples of related problems to learn from. We start by running Twee (without abstractions) on P_h for a fixed amount of time. When it times out, even though it has not proved the goal, it will have derived a large collection of rewrite rules. Each of these derived rules $l \to r$ corresponds to an equation $l = r$ about the theory we are exploring. Many of these equations will be quite boring, but some will represent interesting lemmas. Our idea is to learn abstractions from only the most interesting lemmas.

In order to define interestingness, we first need some auxiliary definitions. For any lemma $l = r$, we shall denote by $T(l = r)$ the set of terms occurring in its proof, including the proofs of any sub-lemmas used in the proof. More precisely, the proof of $l = r$ is a chain $l = t_1 = \ldots = t_n = r$, where each step may be justified by a lemma $l_i = r_i$ having its own proof. We then define $T(l = r) = \{l, t_1, \ldots, t_n, r\} \cup \bigcup_i T(l_i = r_i)$.

For a term t, we define its weight $|t|$ as the number of function applications plus the number of unique variables occurring in t. We extend this to equations by defining $|l = r| = |l| + |r|$, and to sets of terms T by defining $|T| = \sum_{t \in T} |t|$. We then define the "interestingness" $s(l = r)$ of a lemma as follows:

$$s(l = r) := \frac{|T(l = r)|}{|l = r|^2}.$$

The intuition is that lemmas with higher scores are those that are simple statements requiring a long proof. Of course this does not necessarily coincide with

what a human would consider interesting or useful, but we take it as a rough heuristic towards measuring these notions.

Given a partial proof of P_h, we pick out the k lemmas with the greatest interestingness score, $l_1 = r_1, \ldots, l_k = r_k$. We then generate what we call *partial proof abstractions* by running Stitch on $\bigcup_{1 \leq i \leq k} T(l_i = r_i)$. We can then rerun Twee on P_h augmented with these abstractions.

3.2 Domain Abstractions

The goal of this approach is to learn abstractions that are useful for proving problems in a particular domain $\mathcal{D}$. We require a set of problems, some of which are already proved (the "easy" problems) and some of which are not (the "hard" problems). The algorithm works as follows:

1. First, we use Stitch on the proofs of the easy problems to find abstractions for each problem. These abstractions are *local*, i.e. specific to one problem.
2. Then, we augment each problem with its local abstractions and re-run Twee to test if the abstractions are *good*, i.e. allow us to reprove the problem faster.
3. Finally, we build a set of all the good local abstractions, and compress it with Stitch to find commonalities and derive higher-level *domain abstractions*.

This process is illustrated in Fig. 1, with pseudocode in Algorithm 1. There, we use $T(\Gamma(P))$ to denote the set of terms occurring in the proof $\Gamma(P)$ of problem P found by Twee. The set of abstractions derived by running Stitch on $T(\Gamma(P))$ (where $\Gamma(P)$ was found using a specific Twee strategy `strat`) is denoted by $A_P^{\texttt{strat}}$. We use $P \oplus A$ to denote P augmented with abstractions A. Finally, the local abstractions are filtered by an effect metric *em* which measures the speedup obtained by augmenting the problem with the abstraction. More precisely,

$$em(P, A) = \frac{t(\Gamma(P))}{t(\Gamma(P \oplus A))},$$

where $t(\Gamma(P))$ denotes the time taken by Twee to find proof $\Gamma(P)$. Only those abstractions that lead to a speedup above some threshold τ are added to the set of good abstractions.

We see that growing the good abstraction set relies on running the Local-Abstractions subroutine for a collection of different *strategies*. This simply refers to an assignment of specific values to parameters that influence the working of Twee. In particular, the strategy parameters are (1) whether to run Twee with or without its *goal flattening* option [30] and (2) the abstraction weight factor (see Sect. 4.1). By iterating over combinations of these parameters, we increase the pool of good abstractions to derive domain abstractions from later.

We can then use these domain abstractions to augment hard problems in the same domain, with one caveat: some abstractions may mention functions that do not occur in the hard problem, and these are removed.

Algorithm 1.

procedure DomainAbstractions($\mathcal{D}$)
  `good_absts` ← empty set
  for all combinations of $P \in \mathcal{D}$ and `strat` ∈ `strategies` **do**
    $A_P^{\texttt{strat}}, em(P, A_P^{\texttt{strat}}) \leftarrow$ LocalAbstractions(P, `strat`)
    if $em(P, A_P^{\texttt{strat}}) \geq \tau$ **then**
      Add the alpha-normalized $A_P^{\texttt{strat}}$ to `good_absts`
    end if
  end for
  $\mathcal{D} \leftarrow$ Run Stitch over `good_absts`
  Return $\mathcal{D}$
end procedure

Algorithm 2.

procedure LocalAbstractions(P, `strat`)
  Run Twee on P with `strat`.
  If it fails to terminate within the time limit, return failure.
  Else we get proof $\Gamma(P)$.
  Extract terms from $\Gamma(P)$ as strings to get list $T(\Gamma(P))$.
  Run Stitch on $T(\Gamma(P))$ to get abstractions $A_P^{\texttt{strat}}$.
  Run Twee on $P \oplus A_P^{\texttt{strat}}$ with `strat`.
  Compute $em(P, A_P^{\texttt{strat}})$. If Twee timed out on $P \oplus A_P^{\texttt{strat}}$, set $em(P, A_P^{\texttt{strat}}) = 0$.
  Return $A_P^{\texttt{strat}}$, $em(P, A_P^{\texttt{strat}})$.
end procedure

4 Supporting Abstractions in Twee

Twee's architecture is based on *unfailing completion* [5,21], which works by treating the input equations as rewrite rules and then deriving new rules that cannot be proved by forward rewriting. For example, given some axioms about group theory, Twee might orient them into the following rules:

$$e \cdot x \to x \qquad x^{-1} \cdot x \to e \qquad (x \cdot y) \cdot z \to x \cdot (y \cdot z)$$

We can observe that the term $(x^{-1} \cdot x) \cdot y$ has two normal forms:

$$(x^{-1} \cdot x) \cdot y \to e \cdot y \to y \qquad (x^{-1} \cdot x) \cdot y \to x^{-1} \cdot (x \cdot y),$$

hence the equation $x^{-1} \cdot (x \cdot y) = y$ is true but cannot be proved by forward rewriting, since both sides are normal forms (this is called a *critical pair*). Twee therefore adds the new rewrite rule $x^{-1} \cdot (x \cdot y) \to y$. This process of deriving new rewrite rules continues until the rule set is saturated or the goal is proved.

The number of critical pairs is typically quadratic in the number of rewrite rules [16, Section 2.2], and may reach $\sim 10^{10}$ in a longer proof attempt. Hence, rather than turning all critical pairs into rules, Twee repeatedly picks the "best" critical pair and turns it into a rule (following a DISCOUNT loop [4]). To judge

the goodness of a critical pair, Twee first normalizes it with respect to the existing rewrite rules, then computes a score for the resulting equation. The score has several components (see [30]) but the most important one is the equation's *weight*, defined as the number of symbols in the equation:

$$\begin{aligned} w(t = u) &= w(t) + w(u) \\ w(x) &= 1, \text{ if } x \text{ is a variable} \\ w(F(t_1, \ldots, t_n)) &= 1 + \sum_{i=1}^{n} w(t_i), \text{ if } F \text{ is a function symbol} \end{aligned}$$

Our approach to abstractions is inspired by the *weighting strategy* [24]: whenever a term or subterm in a critical pair matches an abstraction, reduce its computed weight. Specifically, we extend w with the following clause:

$$w(A\sigma) = w_A + \sum_{x \in dom(\sigma)} w(x\sigma), \text{ if } A \text{ is an abstraction and } \sigma \text{ is a substitution}$$

Here, w_A is a constant representing the weight of the abstraction. We come back to this in Sect. 4.1, but for now, let us assume that $w_A = 1$.

For example, suppose we are studying two functions *times* and *plus*. To encourage the prover to reason about distributivity, we might give the following abstraction A, representing a term in which distributivity can be applied.

$$plus(times(x, y), times(x, z))$$

Now suppose that the following equation occurs as a critical pair:

$$times(c, \underline{plus(times(g(f(a), b), f(x)), times(g(f(a), b), g(y, z)))}) = g(a, b)$$

Without the abstraction A, this equation would have a weight of 21. With A, the underlined term can be matched as $A\sigma$, where $\sigma = \{x \mapsto g(f(a), b),\ y \mapsto f(x), z \mapsto g(y, z)\}$. Hence the total weight is $1 + w(c) + w(A\sigma) + w(g(a, b)) = 2 + w_A + w(g(f(a), b)) + w(f(x)) + w(g(y, z)) + 3 = 15$. The lesser weight means that this critical pair will be selected sooner.

Note that A reduces the equation's weight in two ways: (1) the abstraction's "skeleton", $plus(times(\_, \_), times(\_, \_))$, only counts for weight 1 instead of 3; (2) although x occurs twice in the abstraction, $x\sigma$ is only counted once, hence the repeated subterm $g(f(a), b)$ is only counted once. Property (2) is vital in order for abstractions to act like *abbreviations*. For example, in Sect. 1.1 we define the abstraction $f(x, x)$, where f is the Sheffer stroke, which represents the negation of x. Property (2) ensures that $f(t, t)$ has the same weight as a hypothetical term $not(t)$.

As an aside, we have also implemented term-level *resonators* [37] in Twee. These work just the same as abstractions, except that they only match if every variable in σ is mapped to a variable (not a compound term).

4.1 The Cost of an Abstraction

Above we assumed that the weight w_A of an abstraction was 1. In fact, we allow the user to choose how w_A is computed. There are two strategies, both parametrized on a user-specified constant k: *(1) Constant weight:* Every abstraction A has the same weight, $w_A = k$. *(2) Weight factor:* We set $w_A = k\, w_{skel}(A)$, where $w_{skel}(A)$ is the weight of A if all variables are counted as weight 0. In the example above, $w_{skel}(A) = 3$. This gives larger abstractions greater weight.

4.2 Implementing Hints using Rewriting

We can implement w efficiently using ideas from term rewriting. Let us call the original weighting function, without the $w(A\sigma)$ clause, w_0. First, for each abstraction $A := t[x_1, \ldots .x_n]$, we introduce a fresh function symbol $A_t(x_1, \ldots, x_n)$. Then, to weigh a critical pair, we first *replace* any term of the form $t[u_1, \ldots, u_n]$ with $A_t(u_1, \ldots, u_n)$, and then weigh the result using w_0. For the example above, if we call the abstraction function A, replacement produces the following term:

$$times(c, A(g(f(a), b), f(x), g(y, z))) = g(a, b)$$

and we can see that its weight according to w_0 is indeed 15. We can see this as *normalizing* the critical pair with respect to the following rewrite rule:

$$plus(times(x, y), times(x, z)) \to A(x, y, z)$$

and this is in fact how Twee implements it. The function w transforms the abstractions into a set of rules like the one above, and normalizes the critical pair with respect to those rules before weighing it.[5] Hence we can use Twee's term indexing machinery to efficiently handle large numbers of abstractions.

We have not specified what happens if a term matches multiple abstractions. This is determined by the particulars of Twee's normalization algorithm. This uses outermost reduction, and prefers more specific to more general matches.

5 TPTP Experiments

We have evaluated Twitch on a set of unsatisfiable unit equality (UEQ) problems from TPTP v9.2.1 [32]. The problems are drawn from nine TPTP domains, summarized in Table 1. The main purpose is to measure whether the abstractions discovered by Twitch help in solving problems from the TPTP.

Section 5.1 measures whether partial proof abstractions and domain abstractions improve Twee's runtime. For domain abstractions, the user must define which problems make up the domain; in our experiments (with one exception

[5] Note that these rules are purely internal to w and are not added to the clause set. The normalized term is thrown away after w is done with it.

mentioned in Sect. 5.2), we considered the domain to be the set of problems in a given TPTP domain (e.g. Group Theory (GRP), Lattice Theory (LAT), etc.)

Section 5.2 reports on the hard problems solved using abstractions from Twitch. We define a problem as hard if it has TPTP rating at least 0.9 (i.e. 90% of ATPs cannot solve it within 300 s) and Twee (without abstractions) cannot solve it within 1000 s. There are 70 such problems from Table 1.

Section 5.3 considers various ablations testing some aspects of Twitch's architecture and strategies.

Table 1. Unsatisfiable UEQ problems per TPTP domain, TPTP v9.2.1

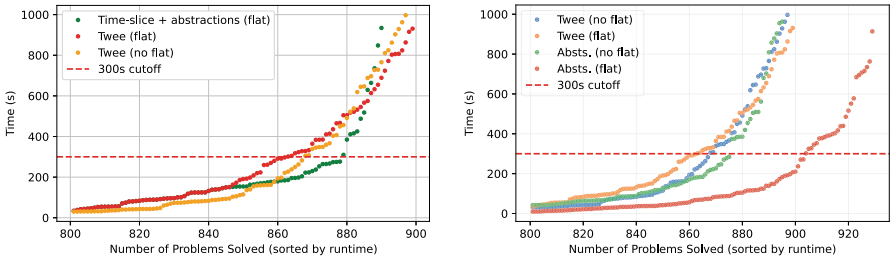
Domain	Count
ALG (General Algebra)	20
BOO (Boolean Algebra)	56
COL (Combinatory Logic)	120
GRP (Group Theory)	481
LAT (Lattice Theory)	126
LCL (Logic Calculi)	84
REL (Relation Algebra)	81
RNG (Ring Theory)	48
ROB (Robbins Algebra)	25
Total	1041

5.1 Overall Runtime Improvements

Figs. 2a and 2b include cactus plots of runtimes of Twee on the TPTP problems listed in Table 1. We benchmark Twee both with and without its *goal flattening* strategy [30], in order to see how this strategy interacts with abstractions. In both figures, the 'Twee (flat)' and 'Twee (no flat)' plots show Twee running without abstractions (with and without goal flattening respectively) for the full time limit of 1000 s. We have plotted a line at 300 s, a typical timeout for ATP benchmarking, but continue the evaluation until 1000 s.

Figure 2a shows the results for partial proof abstractions. Here, we first run Twee (without goal flattening) for 150 s, then if it times out, extract partial proof abstractions from the failed proof, and rerun Twee (with goal flattening) augmented with these abstractions for the remaining time.

Figure 2b shows the results when using domain abstractions. The 'Twee (flat)' and 'Twee (no flat)' show the performance of Twee without abstractions. 'Absts. (flat)' and 'Absts. (no flat)' show the performance of Twee once the abstractions are added. Unlike with partial proof abstractions, there is no time-slicing. The plot does not include the time to learn the abstractions for a given domain, as this is shared between all problems in the domain.



(a) Using partial proof abstractions. (b) Using domain abstractions.

Fig. 2. Cactus plots of Twee runtimes for TPTP problems. A given point in these plots corresponds to a particular problem P from Table 1. The y-value of a point indicates the time taken by Twee to find a proof of P (possibly augmented with Twitch domain abstractions), and the corresponding x-value is the index of P when the problems are sorted in increasing order of runtime.

We see that applying just domain abstractions to a problem results in a modest speedup, but also applying goal flattening leads to a large speedup, with roughly 25 more problems solved inside of 300 s. When comparing only the problems that baseline Twee could solve within 300 s, adding domain abstractions with goal flattening roughly halves the runtime. Hence domain abstractions seem to interact particularly well with goal flattening.

Partial proof abstractions provide some improvement in runtimes, and solve more problems after 300 s, but are not as effective as domain abstractions. However, they are still useful in many cases since they do not require any preprocessing to find the abstractions. They also have the advantage of being available for any problem, even those for which we have no other proofs to learn from.

5.2 Solving Hard Problems

Along with general runtime improvements, we find that adding Twitch abstractions allows us to prove 18 hard problems. As an additional baseline, we ran Vampire 5.0.1 on the 12 rating-1 problems solved by Twitch, using the same 1000 s timeout. Vampire solved only three of them—LCL375-10, LCL304-10, and LCL395-10—and was slower than Twitch in each case; the other nine timed out. We summarise the Twitch results here, and highlight some interesting cases.

Partial Proof Abstractions. Given a hard problem, we run Twee on it for at most 500 s. We then generate abstractions from the top lemmas of the partial proof, and supply these to Twee. This allowed us to solve 11 hard problems – each with a total time limit of 1000 s, including the partial proof run. The majority of these (6 out of the 11) are LCL problems that contain translations of Horn clauses into equational logic (using the technique introduced in [12]). In these cases, we find that partial proof abstractions of the form $ifeq(x, true, y, true)$

are always discovered – encodings of implication in terms of the $ifeq$ function. And through manual testing, we find that these are the abstractions that make the real difference in helping solve the hard problems.

Domain Abstractions. Domain abstractions solved 18 hard problems within 1000 s, including every hard problem solved by partial proof abstractions. For instance, problem `LAT075-1` (Rating: 1.00) was solved in 32 s (without goal flattening) by providing the following domain abstractions:

- $f(A, A)$
- $f(f(B, A), f(B, A))$
- $f(f(A, B), f(A, C))$
- $f(f(D, C), f(B, A))$
- $f(f(f(C, f(C, f(f(A, A), B))), B), A)$

Checking manually, we found that the first and third abstractions were crucial to finding a proof in a reasonable amount of time.

Note that the first three abstractions are semantically meaningful. The first one corresponds to the *not* operator, the second one to *and*, and the third to one side of a kind of self-distributivity law satisfied by the Sheffer stroke.

Further inspection of the full results (supplied in the source repository) shows that the abstraction lists are quite long in three cases. These cases highlight that it is sometimes helpful to skip abstracting over the good abstractions (line 10 of Procedure 1) and simply take $A_{\mathcal{D}}$ to be the set of all good abstractions.

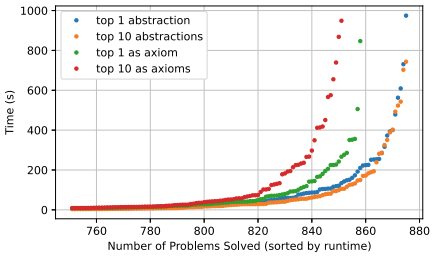
In addition to the 18 problems, Problem `GRP677-1.p` was also solved (in 180 s) using domain abstractions. However, this required constraining the domain $\mathcal{D}$ to only include problems that share very similar axioms to the hard problem being solved. So, we see that a larger domain is not necessarily better. This suggests that, when aiming to solve a particular hard problem, curation of the domain we provide Twitch may be a useful direction for future work.

5.3 Abstractions vs Definitional Axioms

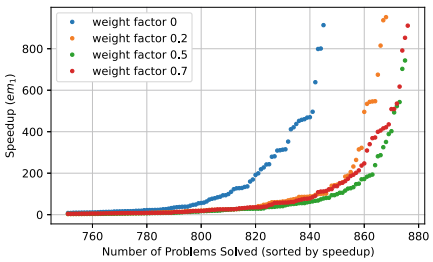
There are two ways one might use Stitch abstractions to augment a problem P – by giving them to Twee as abstractions, or by adding them as definitional axioms (as described in Sect. 1.1). Twitch uses the abstractions approach. We here present experiments that aim to validate this choice, by comparing the two approaches. The experiments involved running Algorithm 2 on the TPTP problems listed in Table 1.

Figure 3a depicts cactus plots of runtimes obtained in line 7 of Algorithm 2 for each problem. The different colours correspond to different approaches in augmenting P with A. The 'top k' in the figure legends refers to taking the top k abstractions found by Stitch in line 6 of Algorithm 2.[6].

[6] Note that we do not necessarily actually use all k abstractions to augment the problem, as some may not be back-translatable (see Sect. 2). Nonetheless, the comparison remains informative as the actual number of abstractions used for a given problem was the same for both the abstraction and axiom approaches.



(a) Cactus plot of runtimes for varying number of abstractions and ways of providing them to Twee – as axioms or abstractions.



(b) Cactus plot of runtimes for varying weight factors (see Section 4).

Fig. 3. The effect of local abstractions. Each point corresponds to a particular problem P from Table 1. The y-value of a point corresponds to the time taken by Twee to find a proof of P when augmented with local abstractions, i.e. abstractions derived from an existing proof of P by default Twee. The x-value corresponds to the index of the problem when the problems are sorted by increasing runtime of Twee P augmented with the local abstractions. In other words, the points correspond to runtimes from line 7 of Algorithm 2.

We see that overall abstractions lead to lower runtimes compared to definitional axioms, and in particular, fewer cases in which Twee times out as a result of the augmentation. As such, the abstractions approach provides a larger pool of good abstractions for Twitch to derive domain abstractions from. Furthermore, we see that increasing from 1 to 10 abstractions leads to a significant increase in timeouts when using axioms, while the abstractions approach is much more robust to this increase. Hence, using the domain abstractions derived from Twitch as Twee abstractions to augment some hard problem allows us to provide more abstractions with a lesser chance of incurring substantial slowdowns.

However, there are still some problems where providing abstractions as axioms lead to more significant speedups compared to using Twee's abstraction mechanism. For example, our earlier motivating `LAT075-1` problem illustrates such a case.[7] This suggests that there may be room for combining the two approaches in future work.

Finally, we include Fig. 3b which shows a cactus plot of abstraction-augmented runtimes with varying weight factors. A weight factor of about 0.5 (halving the cost of terms matching abbreviations) seems to work best. This aligns with the intuition that low weight factors are more aggressive and hence

[7] There are also 15 problems where axioms lead to speedups while abstractions lead to slowdowns. For LAT141-1, we get a proof $\Gamma(P)$ from default Twee (without goal-flattening) in $\sim$ 160 seconds. The top abstraction found by Stitch on this proof is $f(X, Y, Z) = meet(X, join(Y, Z))$. Adding this as an axiom leads to a proof in $\sim$ 12 seconds, while with the abstraction mechanism, it leads to a proof in $\sim$ 260 seconds.

lead to more slowdowns (in particular, timeouts). However, some individual problems have the greatest speed-ups with lower weight factors.

6 Related Work

Our implementation of abstractions in Twee is an automatically controlled instance of the *weighting strategy* [24]. Weighting later influenced the work of Wos on the *resonance strategy* [37]. Resonators are clause patterns that bias clause selection toward shapes deemed promising by a human expert. This approach proved effective in difficult domains [7,14]. At around the same time Veroff introduced the *hint strategy* [33] as well as the *proof sketches* method [34], in which a prover first establishes an easier or related theorem and then reuses clauses from that proof as hints to guide the search for the harder goal. Our abstractions are in spirit the opposite of Veroff's hints: hint clauses are key clauses which might be derived only once, whereas our abstractions are key term structures which appear over and over again.

Our symbolic learning of domain structure in proofs is very related in spirit to the work of Schulz on pattern-learning and term space mapping [13,27,28].

Closely related to (and inspiration for) our use of Stitch is previous work on automated proof compression. Urban et al. [35] propose introducing new definitions to shorten proofs found by saturation provers. Their goal is proof refactoring and compression, whereas we use compression primarily as a mechanism for discovering search-guiding patterns. Similarly, recent work by Wernhard and Zombori [36] explores grammar-based compression of Metamath proofs.

Related to the idea that internal proof structure can be mined to improve future search is work on learning-based guidance for saturation provers, such as ENIGMA [11,17]. However, these approaches rely on statistical models for clause selection, whereas our method extracts explicit syntactic abstractions that provide lightweight and interpretable guidance tailored to equational completion.

The automatic ranking of interesting derived results has been studied by Puzis et al. [26], who use automated reasoning to generate and filter potentially interesting theorems. Their work inspired our simple heuristic for ranking lemmas extracted from partial Twee proofs, and further exploration of their measures in our setting is an interesting direction for future work.

Other approaches to learning from related problems have been explored in automated reasoning. For example, the *Octopus* system [23] combines learning and parallel search, using information from easier problems to guide proofs of harder ones. This aligns with our domain abstraction approach, which is inspired by curriculum learning [8]: simpler problems in a domain provide reusable structural patterns for harder ones.

Automated theorem provers and finite-model finders have long been used as research tools in algebra [20,25]. Recent examples include work on semigroup, ring, loop, and magma varieties, as well as Tao's Equational Theories Project, which used ATPs and Lean to map implications between small equational laws of magmas [2,3,9,15,18,19]. In such settings, the usefulness of ATPs depends on

finding proofs that can be understood and reformulated in standard mathematical terms. Techniques such as proof sketches and proof simplification address this need. Our abstractions are related in spirit: the recurring term patterns identified by Twitch would ideally correspond to algebraically meaningful constructions, and guiding proof search toward these patterns may help produce proofs that are more structured and easier to analyse.

7 Limitations and Future Work

The system we have presented is in many ways quite simple-minded, and there are many ways in which it can be refined. Firstly, our domain construction is quite crude: we restrict to problems within the same TPTP theory sharing concrete symbols, and we do not attempt to automatically generate related or weakened conjectures in the style of proof sketches. Likewise, lemma scoring and term extraction from proofs are based on simple heuristics that could likely be refined as a path to more useful abstractions.

Further, abstraction selection is treated as a single-objective optimization problem focused primarily on runtime speedup. Alternative or multi-objective criteria—such as proof length reduction or robustness across strategies—may yield better guidance. It may also be fruitful to learn abstraction proposals directly from accumulated (problem, abstraction) training data, potentially via language models. Despite these simplifications, we believe our results demonstrate that even a relatively simple abstraction pipeline can produce abstractions yielding substantial improvements and solve previously intractable problems, and that this suggests that automated, structure-based abstraction discovery is a promising direction for future research in equational theorem proving. We also note that for some problems, the best runtimes were obtained by providing both domain and partial proof abstractions. In particular, one more hard problem – `LCL351-10` – was solved this way, where all other approaches failed, indicating potential of further benefits through such combination.

Also, the implementation of abstractions in Twee is currently quite brittle. The problem is that we only give bonuses to critical pairs that contain instances of the abstraction terms. However, Twee demodulates critical pairs, simplifying them using the generated rewrite rules. Thus a term matching an abstraction may well get rewritten to a term that does not match. For example, consider the abstraction $f(x, x)$. If f is also associative ($f(f(x, y), z) \to f(x, f(y, z))$) then the term $f(f(x, x), y)$ (which matches the abstraction) will be rewritten to $f(x, f(x, y))$ (which does not). We are experimenting with a critical pair-like mechanism for generating extra *derived abstractions* in these cases. We can generate many extra abstractions this way, but they look very different from the user-provided abstractions and it is not clear if they respect the user's intent.

Finally, we would like to explore generating not only abstractions but also hints. These are key steps that are needed in a proof, but may be difficult for the prover to find. An example of a good hint might be a rewrite rule which is large (hence is unlikely to be selected by Twee) but leads to a proof of a conjecture or important lemma.

8 Conclusion

We have shown that abstractions, automatically generated via compression of existing proofs, are a promising strategy for equational proving. Augmenting Twee with pre-learned domain abstractions allows us to prove several difficult conjectures and roughly halves the prover's runtime for more typical conjectures.

Creating high-quality abstractions currently requires a supply of simpler problems within the same domain. In cases where these do not exist, we can also generate abstractions from a failed proof attempt, which are currently lower quality. Since typically only a few simple domain abstractions are needed, we hope that a more sophisticated extraction mechanism can solve this problem.

There is clearly much potential to refine the method, both how the abstractions are found and how they are exploited by the prover. Even so, we believe the results are promising and motivate further exploration of this idea.

Acknowledgments. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, and by the Swedish Research Council (VR) grant 2025-06153, Semantically-guided theorem proving for mathematics.

References

1. Twitch Experiment Repository. https://github.com/WeAreDevo/ijcar26-twee_abstractions
2. Araújo, J., Ferreira, F., Janota, M., Kinyon, M.: Forbidden substructure theorems: a computational tool applied to varieties of lazy magmas. Math. Comput. (2026). https://doi.org/10.1090/mcom/4169
3. Araújo, J., Kinyon, M., Robert, Y.: Varieties of regular semigroups with uniquely defined inversion. Port. Math. **76**(2), 205–228 (2019)
4. Avenhaus, J., Denzinger, J., Fuchs, M.: Discount: a system for distributed equational deduction. In: Proceedings of the International Conference on Rewriting Techniques and Applications, pp. 397–402. Springer (1995)
5. Bachmair, L., Dershowitz, N., Plaisted, D.A.: Completion without failure. In: Proceedings of the Rewriting Techniques, pp. 1–30. Academic Press (1989). https://doi.org/10.1016/B978-0-12-046371-8.50007-9
6. Bártek, F., Chvalovský, K., Suda, M.: Regularization in spider-style strategy discovery and schedule construction. In: Benzmüller, C., Heule, M.J., Schmidt, R.A. (eds.) Automated Reasoning, pp. 194–213. Springer Nature Switzerland, Cham (2024)
7. Beeson, M., Wos, L.: Finding proofs in Tarskian geometry. J. Autom. Reason. **58**(1), 181–207 (2017). https://doi.org/10.1016/B978-0-12-046371-8.50007-9
8. Bengio, Y., Louradour, J., Collobert, R., Weston, J.: Curriculum learning. In: Proceedings of the 26th Annual International Conference on Machine Learning, pp. 41–48. ICML '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1553374.1553380
9. Bolan, M., et al.: The equational theories project: advancing collaborative mathematical research at scale. arXiv preprint arXiv:2512.07087 (2025)

10. Bowers, M., et al.: Top-down synthesis for library learning. Proc. ACM Program. Lang. **7**(POPL), 1182–1213 (2023). https://doi.org/10.1145/3571234
11. Chvalovský, K., Jakubův, J., Suda, M., Urban, J.: ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In: Proceedings of the Automated Deduction - CADE 27: 27th International Conference on Automated Deduction, Natal, Brazil, August 27–30, 2019, Proceedings, pp. 197–215. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-030-29436-6_12
12. Claessen, K., Smallbone, N.: Efficient encodings of first-order Horn formulas in equational logic. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) Automated Reasoning, pp. 388–404. Springer, Cham (2018)
13. Denzinger, J., Schulz, S.: Learning domain knowledge to improve theorem proving. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE 1996. LNCS, vol. 1104, pp. 62–76. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61511-3_69
14. Fitelson, B., Wos, L.: Missing proofs found. J. Autom. Reason. **27**(2), 201–225 (2001)
15. Hemmila, D., Phillips, J.D., Rowe, D., White, R.: The varieties of rings of Bol-Moufang type. J. Algebra Appl. **23**(11), 2450177 (2024). https://doi.org/10.1142/S0219498824501779
16. Hillenbrand, T., Löchner, B.: The next Waldmeister loop. In: Voronkov, A., (ed.) Automated Deduction–CADE-18, pp. 486–500. Springer, Heidelberg (2002)
17. Jakubův, J., Urban, J.: Enigma: Efficient learning-based inference guiding machine. In: Geuvers, H., England, M., Hasan, O., Rabe, F., Teschke, O. (eds.) Intelligent Computer Mathematics, pp. 292–302. Springer, Cham (2017)
18. Kinyon, M., Phillips, J.: Loops with Squares in Two Nuclei (2025). arXiv preprint arXiv:2510.19961
19. Kinyon, M., Stanovskỳ, D.: Abelianness and centrality in inverse semigroups. Proc. Edinb. Math. Soc. **69**(1), 160–176 (2026)
20. Kinyon, M., Phillips, J.: Rectangular quasigroups and rectangular loops. Comput. Math. Appl. **49**(11), 1679–1685 (2005). https://doi.org/10.1016/j.camwa.2005.01.018. https://www.sciencedirect.com/science/article/pii/S0898122105001896
21. Knuth, D.E., Bendix, P.B.: Simple Word Problems in Universal Algebras, pp. 342–376. Springer, Heidelberg (1983). https://doi.org/10.1007/978-3-642-81955-1_23
22. McCune, W.: OTTER 3.3 reference manual. Tech. rep. (2003)
23. Newborn, M., Wang, Z.: Octopus: combining learning and parallel search. J. Autom. Reason. **33**(2), 171–218 (2004)
24. Overbeek, R., McCharen, J., Wos, L.: Complexity and related enhancements for automated theorem-proving programs. Comput. Math. Appl. **2**(1), 1–16 (1976). https://doi.org/10.1016/0898-1221(76)90002-X. https://www.sciencedirect.com/science/article/pii/089812217690002X
25. Phillips, J.D., Stanovský, D.: Automated theorem proving in quasigroup and loop theory. AI Commun. **23**(2–3), 267–283 (2010)
26. Puzis, Y., Gao, Y., Sutcliffe, G.: Automated generation of interesting theorems. FLAIRS, pp. 49–54 (2006)
27. Schulz, S.: Term Space Mapping for DISCOUNT. In: Denzinger, J., Spencer, B. (eds.) Proceedings of the CADE-15 Workshop on Using AI methods in Deduction, Lindau (1998)
28. Schulz, S., Brandt, F.: Using term space maps to capture search control knowledge in equational theorem proving. In: Kumar, A.N., Russell, I., (eds.) Proceedings of the 12th FLAIRS, pp. 244–248. AAAI Press, Orlando (1999)

29. Schulz, S., Möhrmann, M.: Performance of clause selection heuristics for saturation-based theorem proving. In: Proceedings of the International Joint Conference on Automated Reasoning, pp. 330–345. Springer (2016)
30. Smallbone, N.: Twee: An Equational Theorem Prover, pp. 602–613. CADE (2021)
31. Sutcliffe, G.: The TPTP Problem Library. https://tptp.org/UserDocs/ProblemLibraryManual/TPTPTR.shtml (2025)
32. Sutcliffe, G., Suttner, C.: The TPTP problem library. J. Autom. Reason. **21**(2), 177–203 (1998)
33. Veroff, R.: Using hints to increase the effectiveness of an automated reasoning program: case studies. J. Autom. Reason. **16**(3), 223–239 (1996)
34. Veroff, R.: Solving open questions and other challenge problems using proof sketches. J. Autom. Reason. **27**(2), 157–174 (2001)
35. Vyskočil, J., Stanovskỳ, D., Urban, J.: Automated proof compression by invention of new definitions. In: Proceedings of the International Conference on Logic for Programming Artificial Intelligence and Reasoning, pp. 447–462. Springer (2010)
36. Wernhard, C., Zombori, Z.: Mathematical knowledge bases as grammar-compressed proof terms: exploring metamath proof structures (2025). https://arxiv.org/abs/2505.12305
37. Wos, L.: The resonance strategy. Comput. Math. Appl. **29**(2), 133–178 (1995). https://doi.org/10.1016/0898-1221(94)00220-F. https://www.sciencedirect.com/science/article/pii/089812219400220F