Tactical Diagrammatic Reasoning*

Sven Linker
Department of Computer Science
University of Liverpool, UK
s.linker@liverpool.ac.uk

Jim Burton
Visual Modelling Group
University of Brighton, UK
j.burton@brighton.ac.uk

Mateja Jamnik
Computer Laboratory
University of Cambridge, UK
Mateja.Jamnik@cl.cam.ac.uk

Although automated reasoning with diagrams has been possible for some years, tools for diagrammatic reasoning are generally much less sophisticated than their sentential cousins. The tasks of exploring levels of automation and abstraction in the construction of proofs and of providing explanations of solutions expressed in the proofs remain to be addressed. In this paper we take an interactive proof assistant for Euler diagrams, Speedith, and add tactics to its reasoning engine, providing a level of automation in the construction of proofs. By adding tactics to Speedith's repertoire of inferences, we ease the interaction between the user and the system and capture a higher level explanation of the essence of the proof. We analysed the design options for tactics by using metrics which relate to human readability, such as the number of inferences and the amount of clutter present in diagrams. Thus, in contrast to the normal case with sentential tactics, our tactics are designed to not only prove the theorem, but also to support explanation.

1 Introduction

Automated and interactive reasoning with heterogeneous and purely diagrammatic systems has been possible since the 1990s. Hyperproof [3], Openproof [2], Diabelli [18] and MixR [19] are prominent examples of heterogeneous systems, whilst DIAMOND [8] and EDITH [16] demonstrate differing purely diagrammatic approaches. Speedith [20] is a more recent interactive theorem prover for spider diagrams, an Euler-based notation equivalent to monadic first order logic with equality [7]. None of these tools make use of tactics. Tactical theorem provers enable the strategic application of sequences of inference rules, providing a form of semi-automated theorem proving.

As part of an investigation into the readability of diagrammatic proofs[1], we implemented the first tactical diagrammatic theorem prover by extending Speedith[2]. Our aims were to provide users with convenient, high level ways to prove theorems and to enable tactics to be used as a type of explanation or description of their proof strategies. The design of our tactics is informed by our readability investigation, and so we prioritise the presentation and inspection of steps applied by a tactic. Thus, we make efforts to ensure that the sequence of inferences involved in the tactic are coherent for human readers. We do this by sequencing inferential steps in ways which are, arguably, similar to the ways in which a human might approach the problem (see Section 5), and which result in diagrams with favourable properties such as containing less syntax than diagrams produced by alternative sequences.

In Sections 2 and 3 we provide context for the work by explaining the recent history of tools for diagrammatic reasoning. Then in Section 4 we describe our tactics and their implementation. In Section 5 we evaluate our work by the use of metrics relating to the readability concerns; these metrics include

*This work is supported by EPSRC grant EP/M011763/1.

[1] http://readableproofs.org

[2] Our extended version of Speedith, alongside a collection of theorems and proofs, is available from http://readableproofs.org/speedith

Autexier, S. and Quaresma, P. (Eds.): Proceedings of 12th International Workshop on User Interfaces for Theorem Provers (UITP 2016)
EPTCS 239, 2017, pp. 29–42, doi:10.4204/EPTCS.239.3

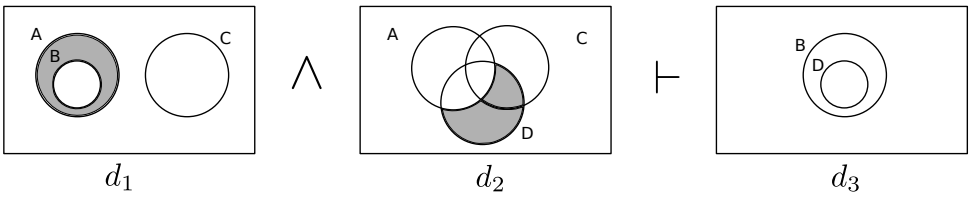


Figure 1: An Euler Theorem

measures such as the length of the resulting proof, the amount of "clutter" or redundant syntax present in each diagram, and so on. Clutter is relevant because several empirical studies have shown that it has a negative impact on user comprehension (see, for example, [10]). Finally, we conclude by describing our goals for the future which include a plan to augment Speedith further with fully automated reasoning and to use it in educational settings to teach diagrammatic reasoning.

2 Background

Diagrammatic logics such as Euler diagrams have a number of striking differences from sentential logics. One of the most important of these is the fact that diagrams can reveal certain consequences of logical statements that would require several inferential steps in sentential systems. This property is known as a "free ride" [15]. In addition, empirical studies have found that logical diagrams can be more accessible to users without training [12], and thus have the potential to bring formal reasoning to wider audiences.

Our work focuses on reasoning with Euler diagrams, a simple and well-known formalism in which the properties of a diagram (the containment, overlap or separateness of circles) directly reflects the underlying meaning (the subsumption, intersection and disjointness of sets). This correspondence between the concrete syntax of diagrams and the semantic level is known as *well matchedness* [4] or, in the terminology of the philosopher C. S. Peirce [1, p126], *iconicity*. Figure 1 shows three unitary diagrams (that is, ones which do not contain sentential logical operators). Unitary diagrams can be joined by conjunction and disjunction to form compound diagrams, and may also be negated by drawing a horizontal bar above a unitary diagram; in this work we deal only with unitary and conjunctive compound diagrams. In Figure 1, diagram d_1 contains three *contours* labelled A, B and C representing sets. From the placement of the circles we can see that the *region* $A \cap C$ is not drawn in d_1: this means that $A \cap C = \emptyset$, since missing regions represent the empty set. A *shaded* region means that the set it represents is empty. For example, from the shading inside A but outside of B in d_1 we see that the set $A - B$ is empty. Each diagram contains a number of *zones*, which are determined by the set of contours they are inside. Thus, there is a zone outside of all contours, one inside A but outside B and C, one inside B only, and one inside C only: therefore, d_1 contains 4 zones. A diagram which contains all possible zones for a given set of contours (that is, all possible overlaps of the contours are drawn) is said to be in *Venn form* (for example, diagram d_2 in Figure 1). If a diagram does not contain all possible zones, the zones that are not drawn are *missing*. Observe that the emptiness of an intersection can be denoted by shading a given zone, or by not drawing it at all. Inference rules for Euler diagrams carry out logically valid operations such as adding and removing contours and shaded zones, and combining the information from several diagrams. In Figure 1, d_3 is a consequence of $d_1 \wedge d_2$.

The first automated reasoner for Euler diagrams was EDITH [16], which came equipped with different sound and complete sets of rules and carried out an A* proof search guided by heuristics [5]. EDITH found the shortest proof for a theorem, given a particular set of rules and provided that a proof exists.

Euler diagrams are the basis for many more expressive notations, such as spider diagrams [7]. Speedith [20] was implemented as an interactive proof assistant for spider diagrams. Speedith makes use of iCircles [17], a Java library which draws Euler diagrams using only circles, and adds the functionality needed to represent spiders (existential elements). Although our work is concerned only with Euler diagrams, we chose Speedith as the basis for extension due to its code base with effective visualisation of all Euler diagrams.

Sentential tactical reasoning, as found in tools such as Isabelle [14], is a powerful semi-automation feature which applies a specific sequence of inference rules. It is sometimes designed to reflect the theorem proving strategies used by humans. Tactics may be used in conjunction with individual inference steps, for instance, in order to simplify the problem before addressing individual cases, or used as high level strategies which may construct the entire proof. The steps taken by a sentential tactic are not normally displayed to the user, but may be made visible if required (e.g., Coq uses a command called `Info` for this). When those steps are displayed, they may be too low level to provide any insight to the user, since sentential tactics are not generally designed with human readers in mind. Since our overall goal is to design perspicuous diagrammatic proofs, and since clarity is one of the key motivations for diagrammatic reasoning in the first place, our tactics are designed with the expectation that users will trace the results of their application.

3 Speedith

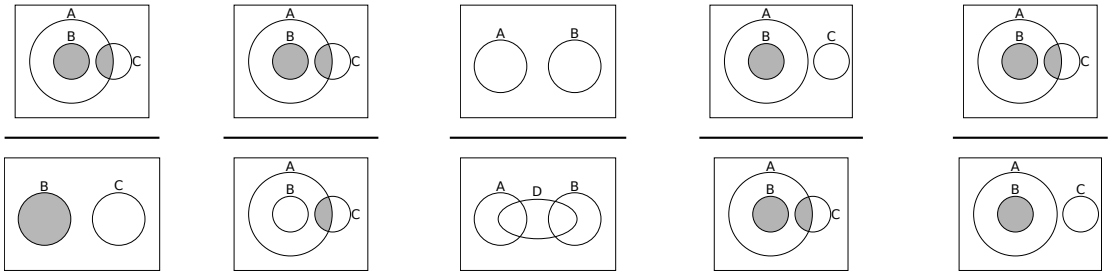
Speedith consists of a reasoning kernel, which can be used to apply rules to the abstract syntax of a given diagram, and a user interface based on iCircles to draw the diagrams. In the following, we will use the term *subgoal* to refer to a compound diagram within a proof of Speedith. A *proof state* is a list of subgoals that still have to be solved. Finally, a *proof* in Speedith is a list of proof states, $g_0, \ldots, g_n$, where for all $0 \leq i < n$, there is a subgoal d in g_i and d' in g_{i+1} such that d' is the result of applying one of the inference rules to d.

The rules can be interactively applied to a subgoal. The position of a subgoal within its proof state is called its *index*. Speedith applies rules backwards; the reasoner tries to reduce a subgoal to the empty diagram, which is a tautology. Hence, we call a proof, $p = g_0, \ldots, g_n$, *finished* if g_n is the empty diagram.

The set of rules within Speedith can be divided into *purely logical rules* and *diagrammatic rules*. Rules in the former set directly correspond to rules and equivalences of propositional logic, while rules in the latter set manipulate the diagrammatic structure of one or more unitary diagrams. In this paper, we are only concerned with the subset of Speedith rules that are applicable to Euler diagrams. Furthermore, we only describe the subset of rules we used in the definition of our tactics. This set is not logically complete, but extending the application to include a complete set would be technically straightforward.

We consider the following diagrammatic inference rules:

1. Erase contour,
2. Erase shading,
3. Introduce contour,
4. Introduce shaded zone,
5. Remove shaded zone,
6. Combine diagrams,
7. Copy contour, and
8. Copy shading.



(1) Erase Contour (2) Erase Shading (3) Intr. Contour (4) Intr. Shaded Zone (5) Rem. Shaded Zone

Figure 2: Examples of Rules 1-5

Rules 1-5 (see Figure 2) only affect single unitary diagrams, while rules 6-8 (see Figure 3) take several unitary diagrams within the premises into account. Furthermore, rules 1 and 2 remove information from the manipulated diagrams, while rules 3-5 are equivalences.

Erase Contour (see Figure 2.1) removes a contour from a unitary diagram. If this contour was separating a shaded zone from a non-shaded zone, then the unified zone in the result will not be shaded. This rule removes information, and so the premiss implies the conclusion but not vice versa. For example, in the premiss of Figure 21, the contour A separated the two zones within contour C. Since one of them is shaded and the other is not, removing A also removes the shading within C.

Erase Shading (see Figure 2.2) removes the shading of a single zone from a unitary diagram. This rule also is not an equivalence rule since it removes information.

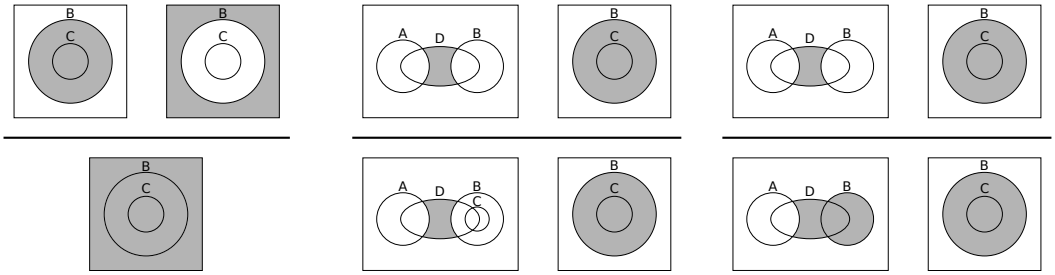
Introduce Contour (see Figure 2.3) is used to add a new contour to a unitary diagram, such that it intersects all visible zones. In this way, the addition of the new contour does not introduce new information into the unitary diagram. This rule is an equivalence rule.

Rules 4 (see Figure 2.4) and 5 (see Figure 2.5) are used to replace a missing zone with a shaded zone within a unitary diagram and vice versa. Both of these rules preserve equivalence.

When using rule 6 (see Figure 3.6) to *combine* two unitary diagrams (also an equivalence), both diagrams must contain the same set of zones. In the result, the two diagrams are replaced by a single diagram with the same set of zones, and in which a zone is shaded if and only if it is shaded in one of the original diagrams.

For the copy rules, 7 and 8, we have to identify which zones in different diagrams *correspond* [6] to each other; that is, which represent the same underlying set. *Copy Contour* (see Figure 3.7) can be used to copy a contour, c, from a unitary diagram d_1 to a second unitary diagram, d_2. The placement of c respects the topological relations specified in d_1 between itself and the contours that occur in both diagrams. The next rule in our system is *Copy Shading* (see Figure 3.8). If a set of zones, z_1, is not shaded in a diagram, d_1, and corresponds to a set of shaded zones, z_2, in a second diagram, then *Copy Shading* can be used to shade all zones within z_1. Consider Figure 3.8. The two zones within contour B in the left premiss and the two zones within B in the right premiss correspond to each other (both denote the whole of B). Hence we can apply *Copy Shading* to shade two zones in the left diagram. Both of these rules are equivalence rules.

The only purely logical rule we consider for tactical reasoning is *idempotency*. It may be used to remove one of two identical conjuncts. So, from $d \wedge d$ we get simply d.



(6) Combine (7) Copy Contour (8) Copy Shading

Figure 3: Examples of Rules 6-8

Table 1: Tacticals of Speedith

Name	Type
THEN	$Tactic \to Tactic \to Tactic$
ORELSE	$Tactic \to Tactic \to Tactic$
REPEAT	$Tactic \to Tactic$
COND	$GoalPredicate \to Tactic \to Tactic \to Tactic$
DEPTH_FIRST	$GoalPredicate \to Tactic \to Tactic$
id	*Tactic*
fail	*Tactic*

4 Tactics

Here, we give an overview of the implemented tactics and their restrictions. The implementation of tactics was originally inspired by the Isabelle interactive symbolic theorem prover, but differs in a number of ways: the choice of programming language is Scala [13], the features are distinctive to diagrammatic reasoning, and our goal is for users to be able to inspect the proof. In sentential reasoners, the user is typically shown only the remaining subgoals after applying a tactic. In contrast, we include a feature within Speedith to display a full proof including all inference rule applications.

Each tactic is of the type $Goals \to Int \to TacticApplicationResult \to Option[TacticApplicationResult]$. It takes as parameters the current proof state (of the type *Goals*) and the index of the subgoal to which the tactic will be applied. The final parameter, of the type *TacticApplicationResult*, contains the list of rules that were applied by tactics during the current invocation of a tactic by the user and the proof state after applying these rules. This result will only be updated by rule-level tactics (see Section 4.1). The type $Option[A]$ in Scala corresponds to the type of the same name in ML [11]. It consists of two constructors *Some* and *None*, where the first takes a parameter of type A. Since the application of a tactic may fail, it can either return a new *TacticApplicationResult* (the updated list of rule applications) or nothing. The tactics can only be applied to a subgoal of the form $D \to D'$, where the consequent D' is a single unitary diagram and the antecedent D is a conjunction of several unitary diagrams. This is a deliberate restriction for simplicity.

To combine tactics and build complex tactics from simpler ones, we implemented combinators or *tacticals* in the fashion of Isabelle: see Table 1, where the second column shows the functional type of each tactical.

The combinator *THEN* takes two tactics as parameters, executes the first, and subsequently executes the second. If either tactic fails, so does the tactic formed by their combination. *ORELSE*, on the other hand, executes the second tactic only if the first tactic fails. Hence the combination only fails if both tactics fail to execute on the given subgoal. A *GoalPredicate* is a function from the proof state (a list of subgoals) and the index of a subgoal to *Bool*. In the tactic *COND* the argument of type *GoalPredicate* is used to analyse the current proof state and choose whether the first or the second argument should be executed. For repeating tactics we employ two tacticals. *REPEAT* applies the given tactic until it fails. For a more guided search, *DEPTH_FIRST* should be used. It repeats a tactic until the predicate given as a first parameter returns true. The tactics *id* and *fail* always succeed or fail, respectively.

The tactics do not increase the expressiveness of Speedith. For the sake of this description, we classify the tactics we implemented into three categories of increasing complexity: *rule-level*, *low-level* and *high-level*. Rule-level tactics apply a single rule once. Low-level tactics combine rule-level tactics to manipulate the premises in a certain way. For instance, Tactic 1, described in Section 4.2 below, converts the unitary diagrams in the antecedent into Venn diagrams, that is, diagrams without missing zones. Finally, high-level tactics may use both rule-level and low-level tactics to implement a particular reasoning strategy.

Currently, users can choose a tactic to be applied from a menu, similar to the way single rules are chosen. By default, only the high-level tactics are shown in the menu, but users can change the preferences to show also the low-level tactics. Since rule-level tactics are simply applications of single rules, they do not appear in the tactics menu.

4.1 Rule-Level Tactics

There is no notion of higher-order resolution for diagrams. Thus, we implemented a tactic for each individual rule, that is, each rule-level tactic applies exactly one of the rules. In that way, soundness of the tactics is dependent on the soundness of the implementation of the set of rules. Most of the rules need additional arguments. For example, *Erase Contour* must be supplied with the unitary diagram within the antecedent to which it will be applied and the name of the contour to be erased. To achieve this, we employ two types of functions: *diagram predicates* and *choosers* (see Table 2). The predicate returns *True* if the accompanying tactic can be applied to a given diagram. The chooser function is applied to the diagram identified by the predicate to identify the diagrammatic element required by the tactic. For example, a suitable predicate for the tactic which applies *Erase Contour* could return *True* if the given diagram is unitary and contains a contour. A suitable chooser function would return an element from the set of contours; depending on the context, this might be an arbitrary contour or the chooser function could use a more complex implementation to choose a candidate that meets certain requirements. The *Option* datatype allows us to indicate that a chooser function found a suitable element a of type A by using $Some(a)$ as the return value, or that no such element exists by using *None*. Internally, a rule-level tactic traverses the syntax tree of the subgoal identified by the supplied index and proof state until the diagram predicate evaluates to *True*, yielding the diagram d. Then it applies the given chooser function to d to get the final parameter. With this information, it can apply the rule it implements and update the result of the tactic accordingly. If any of these steps fail (i.e., the diagram predicate does not evaluate to *True* on any subdiagram; the chooser function does not find a suitable element and returns *None*; or the application of the rule fails), the tactic returns *None*. Otherwise, the return value is the updated list of rule applications, including the new proof state.

Table 2: Auxiliary Functions

Name	Type
DiagramPredicate	$Diagrams \to Bool$
Chooser	$Diagrams \to Option[A]$
GoalPredicate	$Goals \to Int \to Bool$

4.2 Low-Level Tactics

Low-level tactics generally try to apply a single rule as often as possible. In some cases, however, we use additional rules to increase the effectiveness of the tactic, where increased effectiveness could mean less clutter in the resulting diagrams, for instance. The tactics in this section are essentially stepping stones towards the high-level tactics in Section 4.3. Recall that all subgoals are of the form $D \to D'$, where the antecedent D consists of a conjunction of several unitary diagrams, and the consequent D' is unitary. All tactics will try to remove the subgoal whenever it consists of a trivial implication, that is, a subgoal of the form $A \to A$.

The following tactics will be used by higher-level tactics to implement "Venn-style" reasoning. This proceeds in three stages: the diagrams in the antecedent are transformed into Venn-form by introducing all missing zones; contours are introduced to equalise the contours present in each diagram; and finally, the diagrams in the antecedent are iteratively combined into a single unitary diagram. We created two versions of the introduction tactics to implement two versions of Venn-style reasoning: breadth-first and depth-first (cf. Section 4.3). The effect of these tactics is illustrated in Figure 4. In this figure and in all of the following examples in this section, we omit the consequent since the tactics only affect the antecedent.

Tactic 1 (Introduce All Shaded Zones)**.** This tactic (see Figure 4) introduces all missing zones in all unitary diagrams in the antecedent. Hence, it transforms the diagrams into Venn form. It uses rule 4.

Tactic 2 (Introduce All Shaded Zones (Deepest))**.** This tactic (see Figure 4) is similar to tactic 1, but only introduces shaded zones in a conjunction of two unitary diagrams. That is, the tactic "drills down" in the structure of the compound diagram to find and apply itself to a pair of unitary diagrams in conjunction.

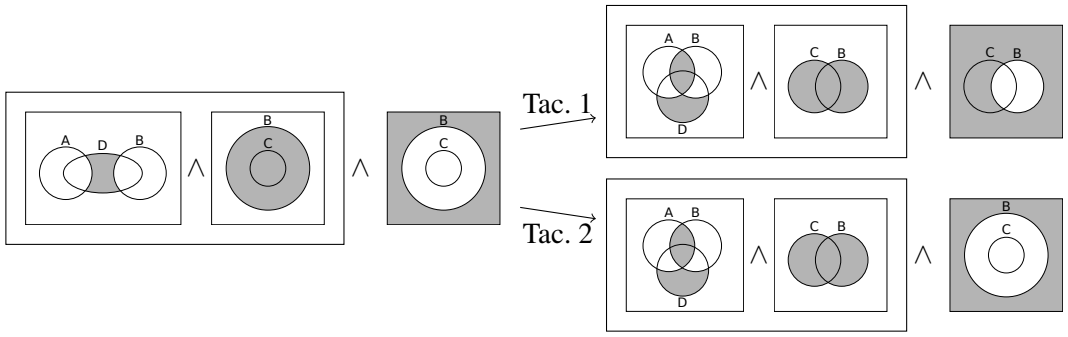


Figure 4: Examples for Introduce All Shaded Zones

Tactic 3 (Introduce All Contours)**.** This tactic (see Figure 5) computes the union, C, of the contour sets in the antecedent. Then, for each unitary diagram, d, it introduces the contours from C that are not present in d. This tactic uses rule 3.

Tactic 4 (Introduce All Contours (Deepest))**.** Similarly to the relationship between tactics 1 and 2, this tactic (see Figure 5) searches for a conjunction of two unitary diagrams and then behaves in the same way as tactic 3. That is, it computes the union of contour sets and introduces the missing contours into the diagrams.

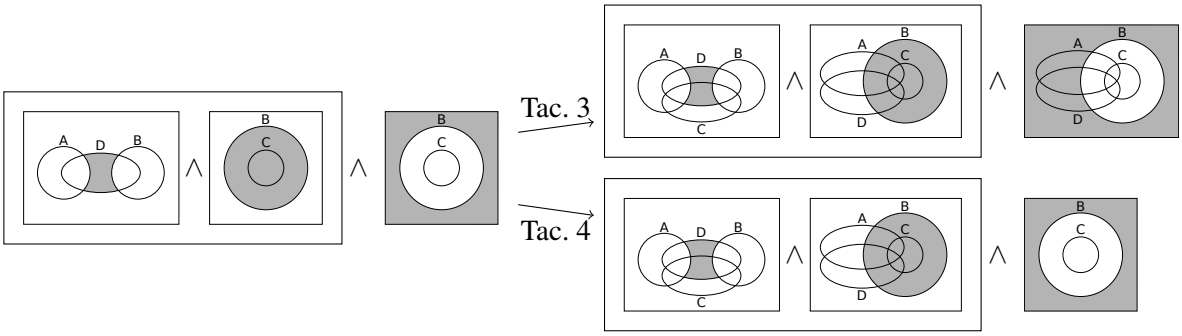


Figure 5: Examples for Introduce All Contours

Tactic 5 (Combine All Diagrams)**.** This tactic searches for conjunctions of unitary diagrams which share the same set of zones, and combines each conjunction into a single unitary diagram. This is done iteratively; if the result of the combination is in conjunction with a suitable diagram, those diagrams will also be combined. This tactic uses rule 6.

The next two tactics are used within high-level tactics to copy elements between conjunctions of unitary diagrams. They provide a more refined way to introduce and remove zones within diagrams.

Tactic 6 (Prepare for Copy Shading)**.** This specialised tactic introduces new shaded zones into diagrams that are part of a conjunction, and between which shading can be copied. It uses rule 4.

Tactic 7 (Prepare For Copy Contours)**.** Similar to tactic 6, this tactic identifies a suitable conjunction and removes shaded zones from both conjuncts using rule 5.

The last low-level tactic is intended to be used as the last step of a derivation, when the antecedent consists of a single unitary diagram that contains all the information needed to reach the consequent. This tactic tries to add and remove elements to transform the antecedent into the consequent.

Tactic 8 (Match Conclusion)**.** This tactic begins by computing the contour sets, A and C, of the antecedent and consequent respectively. It then introduces each contour $c \in C \setminus A$ into the diagrams within the antecedent and subsequently erases each contour $c \in A \setminus C$ from these diagrams. Next, the tactic compares the zones present in the antecedent and the consequent, and introduces zones that are missing in the antecedent but present in the consequent. It then erases shading from zones that are shaded in the antecedent and not shaded in the consequent. Finally, it transforms all zones that are still shaded in the antecedent and not present in the consequent into missing zones. To achieve this, the tactic uses rules 1, 2, 3, 4 and 5.

4.3 High-Level Tactics

The tactics of this section use one or more rule-level or low-level tactics to create more powerful tactics. Some of these tactics are powerful enough to solve all subgoals where the conclusion is a unitary diagram.

The following two tactics copy elements within a conjunction of unitary diagrams and try to reduce the conjunction to a single unitary diagram. They can solve a subgoal as long as only contours or shadings have to be copied.

Tactic 9 (Copy Contours)**.** This tactic begins by identifying a conjunction in which a contour can be copied from one conjunct to the other. It then transforms the shaded zones in this diagram into missing zones to optimise the effect of the copy process. Subsequently, the tactic copies all contours that can be copied between the diagrams. Furthermore, if the conjunction consists of two identical diagrams, it uses idempotency to remove one of the conjuncts. Hence, it uses rule 7, idempotency and tactic 7.

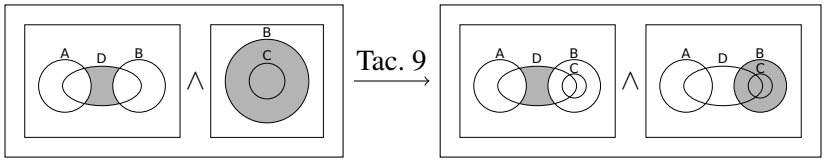


Figure 6: Example for Copy Contours

An example of applying this tactic is shown in Figure 6. The contours are copied between the two diagrams so that the topological relations between the contours are preserved. Observe that the shading information is not taken into account and is not transferred between the diagrams.

Tactic 10 (Propagate Shading)**.** This tactic works similarly to tactic 9 in that it identifies a suitable conjunction to copy shading information, then transforms missing zones to shaded zones and ends by copying the information between the conjuncts. In the conversion phase, it groups the missing zones according to the contours they include and then introduces one of these groups. This tactic uses rule 8, idempotency and tactic 6.

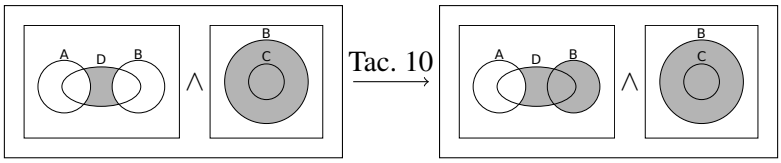


Figure 7: Example for Propagate Shading

Figure 7 shows an example of applying *Propagate Shading*. The tactic does not change the contours present in either diagram. Hence, only the shading of B can be transferred. The shading within the left diagram can not be propagated to the right diagram, since no region corresponds to this shaded zone.

The next two tactics implement "Venn-style" reasoning. They transform the diagrams in the antecedent into Venn-form, combine them until only a unitary diagram remains, and then try to match the antecedent with the consequent. The difference between the two tactics is the order in which the steps are performed. In the breadth-first approach, all unitary diagrams in the antecedent are transformed into Venn-form before any diagrams are combined. Within the depth-first approach, only a single conjunction of unitary diagrams is transformed into Venn-form, which is then combined to a unitary diagram, a process which is repeated until only a single unitary diagram remains.

Tactic 11 (Venn (Breadth))**.** This tactic (see Figure 8) employs several other tactics to solve a subgoal in the following way. First, it converts all unitary diagrams in the antecedent into Venn diagrams by

introducing missing zones. It then adds contours to the resulting diagrams until they all contain the same set of zones, though shading within the diagrams may differ. The tactic then combines conjunctions until only a single unitary diagram remains using the *REPEAT* tactical – note that combining diagrams will fail if only one diagram remains. This diagram is then matched with the diagram in the consequent. This tactic uses tactics 1, 3, 5 and 8.

Tactic 12 (Venn (Depth))**.** The idea behind this tactic is similar to tactic 11, in that it also creates Venn diagrams then combines them. However, this tactic (see Figure 8) chooses and works on one of the *innermost* conjunctions of unitary diagrams. When this conjunction is combined to a single unitary diagram, the process starts from the beginning by selecting a new innermost conjunction, until only one unitary diagram remains, using *DEPTH_FIRST* search. Finally, this diagram is matched against the consequent. This tactic uses tactics 2, 4, 5 and 8.

The last high-level tactic combines the tactics for copying elements. It applies both copying tactics 9 and 10 until it reaches the conclusion.

Tactic 13 (Copy Shading And Contours)**.** This tactic (see Figure 8) uses the copy tactics to collect all information from the antecedent within one unitary diagram. It applies tactic 10 and then tactic 9 until only a unitary diagram remains (note that both of these tactics seek to reduce the number of unitary diagrams in the antecedent by using idempotency). Again, this tactic uses *DEPTH_FIRST* to check when the result is unitary. The resulting diagram is matched to the consequent using tactic 8.

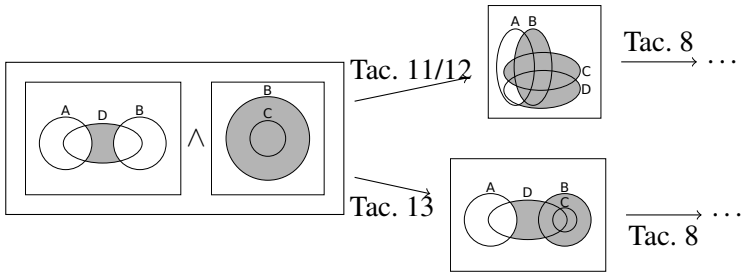
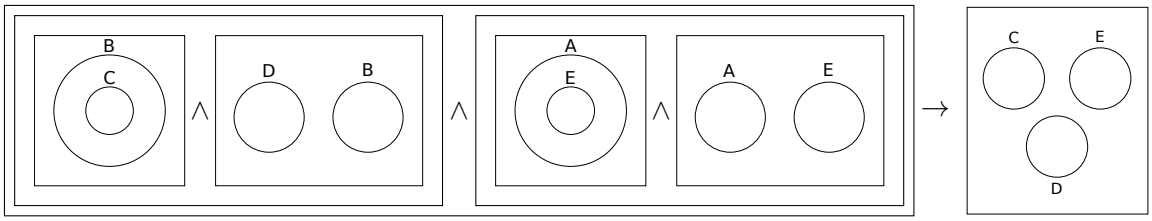


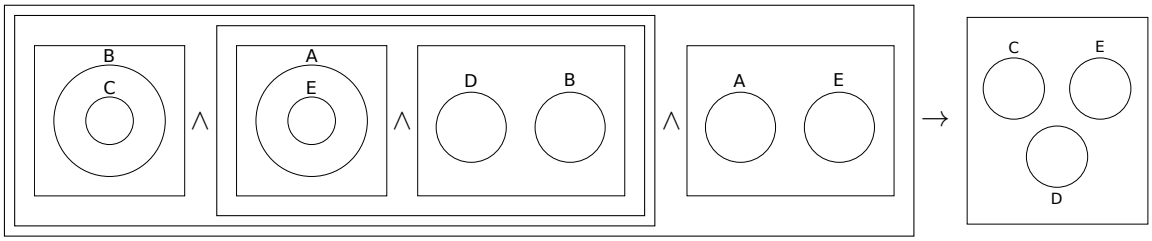
Figure 8: Examples for Venn-Style and Copy Tactic

Figure 8 shows the state during the execution of either of the Venn-style tactics and of the Copy tactic, right before the tactics try to match the antecedent to the consequent. The Venn-style tactics create a Venn-diagram, where all the information of the antecedent is visible through the shading. In contrast, in the case of the Copy tactic, part of this information is visualised through the topological relations between the contours. Both diagrams contain the same semantic information.

Of course, these are not the only possible high-level tactics one could define. We chose this set of tactics for the following reasons. Venn-style reasoning has been traditionally used for Euler-based diagram languages. For example, in the original definition of Spider diagrams [7], the only rule usable for Euler diagrams to transfer information from one diagram to the other is *Combine*, which requires diagrams with the same set of zones as input. Hence Venn-style reasoning is the only way to join the information contained within several diagrams. However, Speedith [20] allows for the use of the copy rules, and we have found the copying tactics provide effective reasoning strategies in many situations. Furthermore, we do not allow for user-defined tactics. That is, in order to create new tactics in addition to the ones described above, users would have to write Scala code directly. This is necessary because writing a new tactic requires knowledge of the implementation of the individual rules and tactics on which the new tactic would depend.



(a) Flat Conjunctive Structure



(b) Deep Conjunctive Structure

Figure 9: Structural Differences in Theorems

5 Evaluation

The previous version of Speedith functioned as a simple proof assistant, enabling the application of individual rules. In our extension, the tool can still be used in the traditional way, and our tactical reasoning features appear in a separate menu. We also added a number of more minor enhancements related to usability, such as the ability to save and load proofs, and an undo capability which allows the user to return to any subgoal by discarding the subsequent steps.

We carried out an empirical study on the effects of clutter on readability of diagrammatic proofs, which showed that participants took significantly longer to identify rule applications in cluttered diagrams [10]. Thus, we use various metrics about the readability of proofs to guide the choices of tactics used in proofs. In our version of the tool, we calculated the following metrics for proofs: the length of the proof, the total and average clutter of the proof (where clutter is calculated as the number of all zones present plus the number of shaded zones) and the maximum *clutter velocity* (the largest change in clutter between one subgoal and its subsequent subgoal). These metrics are available through the interface and we used them to compare alternative proofs and tactics.

In sentential tactical reasoning it is normally the case that powerful auto-style tactics are best used when the main concern is that the reasoner finds a proof, whether or not the user wants to inspect or understand that proof. Our tactics 11 and 12, for Venn-style reasoning, can be considered auto-style tactics, and will frequently reach a proof without further input from the user. However, an important difference is that tactic 12 is designed to work in a way which is perhaps closer to the way a human might attempt to apply the same strategy. The approach taken by the breadth first version (tactic 11) is effective from a technical point of view but produces relatively long proofs containing cluttered diagrams. Arguably, a more likely human approach would be to maintain focus on one or two unitary diagrams at a time: add contours and shaded zones to two unitary diagrams until they have the same zone set, then combine, then pick another two diagrams to work on, and so on. Our design of tactic 12, which maintains focus on two diagrams at a time, relates to the "story telling" perspective of making and reading proofs, and to their communicative rather than their formal function. Whether this tactic really does match

the approach taken by humans remains to be proven empirically. However, in many cases the depth first approach also produces sequences of inferences which are shorter and which contain less cluttered diagrams. Thus, relative to tactic 11, tactic 12 produces proofs which, according to the results of our empirical study, will frequently be easier for readers to understand. Although we don't have room to display entire proofs in this work[3], we measured the performance of the different approaches when applied to the same theorem. The theorem shown in Figure 9a can be proved by either of the Venn-style tactics, and the metrics for each proof is shown in Table 3, where overall clutter is the clutter of the entire proof, showing better performance for the depth first approach.

The behaviour of tactics is heavily reliant on the concrete structure of the theorem to prove. Consider the examples in Figure 9a and 9b. Table 4 shows the metrics of proofs obtained by using tactic 13 (Copy Shading and Contours) to each diagram, where maximal clutter is the sum of the clutter in the most cluttered proof step. Here we see a marked difference in the metrics, and our study predicts that the proof of the theorem in Figure 9a would be significantly easier for a user to understand, relative to the proof of the equivalent theorem in Figure 9b. This difference is mostly due to the fact that in Figure 9a each of the inner conjunctions allows for the application of exactly one of the copy tactics: the left conjunct only allows for the application of *Copy Contours* (tactic 9), while the right conjunct can only be reduced by applying *Propagate Shading* (tactic 10). Subsequently, the remaining contours can be copied between the resulting diagrams. For Figure 9b, however, all contours need to be copied before anything else is done, so that the resulting conjunction contains all 5 contours. Finally, tactic 10 can be applied to this conjunction, which introduces zones and thus increases the clutter.

Table 3: Two proofs of the theorem in Figures 9a

Tactic	Steps	Overall clutter
Venn (Breadth)	27	355
Venn (Depth)	22	195

Table 4: Applying Tactic 13 to the theorems in Figs 9a and 9b

Theorem	Steps	Maximal clutter
Figure 9a	20	27
Figure 9b	28	201

6 Conclusion

By adding tactical reasoning to Speedith, we produced the first diagrammatic reasoning system with this powerful form of semi-automation. We did so in a way that seeks to exploit the distinctively diagrammatic features of the logic, and which takes advantage of what is known about the cognitive benefits and shortcomings of Euler-based notations. We aimed to produce a system that can aid the user to automatically produce proofs which model an approach to problem solving we expect to be more coherent to a human reader and which exhibit features, such as containing relatively less clutter, which we have empirically established as having an positive impact on understanding. Our analysis of tactics and proofs using the metrics described in the previous section enables us to differentiate between several approaches to proving a particular theorem in quantitative ways.

Even though our implementation was inspired by Isabelle, our tactic combinators are typical for systems combining rules in a (semi-) automated fashion. For example, the sequential composition *THEN* and the repeated application *REPEAT* and *DEPTH_FIRST* are similar to constructs of Quantomatic [9]. Our combinators only create single proof states, that is, it is not possible to use backtracking if applying

[3] As stated earlier, a body of proofs is available from our website at `http://readableproofs.org/speedith`

a tactic fails. Extending the implementation to return lists of possible proof states is future work. Furthermore, we intend to implement rules that split conjunctions and disjunctions in appropriate positions into several subgoals. Then, the need for new tactics to split and reduce subgoals naturally arises.

One of our main goals is pedagogical, and we aim to use Speedith and its tactics to teach diagrammatic reasoning. We expect that this type of learning will be supported by allowing the user to experiment with diagrams more directly than is currently possible in Speedith; that is, by drawing, manipulating and experimenting with diagrams themselves. To support this goal, we are working on a Speedith plugin for Openproof, in which Speedith's core reasoning engine and inference rules are made available alongside an Euler diagram editor. Diagrams drawn by the user are converted to the abstractions used by Speedith and checked for validity. In this way, we expect to provide an accessible and novel tool for teachers and students of logic. Finally, although the graphical library we use, iCircles, can draw every Euler diagram, it sometimes produces diagrams with undesirable features such as duplicate labels (whereby a set is represented by two disjoint circles with the same label), and the layout of its diagrams may be suboptimal in other ways. In order to address this, work is underway to improve the layout algorithm of iCircles.

References

[1] A. Atkin (2015): *Peirce (The Routledge Philosophers)*. Routledge.

[2] D. Barker-Plummer, J. Etchemendy, A. Liu, M. Murray & N. Swoboda (2008): *Openproof - A Flexible Framework for Heterogeneous Reasoning*. In: *Diagrammatic Representation and Inference*, *LNAI* 5223, Springer, pp. 347–349, doi:10.1007/978-3-540-87730-1_32.

[3] J. Barwise & J. Etchemendy (1994): *Hyperproof*. CSLI Press.

[4] C. Gurr (1999): *Effective Diagrammatic Communication: Syntactic, Semantic and Pragmatic Issues*. *Journal of Visual Languages & Computing* 10(4), pp. 317–342, doi:10.1006/jvlc.1999.0130.

[5] P. E. Hart, N. J. Nilsson & B. Raphael (1968): *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. *IEEE Transactions on Systems Science and Cybernetics* 4(2), pp. 100–107, doi:10.1109/TSSC.1968.300136.

[6] J. Howse, G. Stapleton, J. Flower & J. Taylor (2002): *Corresponding Regions in Euler Diagrams*. In: *Diagrammatic Representation and Inference*, *LNAI* 2317, Springer, pp. 76–90, doi:10.1007/3-540-46037-3_7.

[7] J. Howse, G. Stapleton & Taylor (2005): *Spider Diagrams*. *LMS Journal of Computation and Mathematics* 8, pp. 145–194, doi:10.1112/S1461157000000942.

[8] M. Jamnik (2001): *Mathematical Reasoning with Diagrams*. CSLI Press.

[9] A. Kissinger & V. Zamdzhiev (2015): *Quantomatic: A Proof Assistant for Diagrammatic Reasoning*. In: *Automated Deduction - CADE*, *LNCS* 9195, Springer, pp. 326–336, doi:10.1007/978-3-319-21401-6_22.

[10] S. Linker, J. Burton & A. Blake (2016): *Measuring User Comprehension of Inference Rules in Euler Diagrams*. In: *Diagrammatic Representation and Inference*, *LNAI* 9781, Springer, pp. 32–39, doi:10.1007/978-3-319-42333-3_3.

[11] R. Milner, M. Tofte & D. Macqueen (1997): *The Definition of Standard ML*. MIT Press, Cambridge, USA.

[12] K. Mineshima, Y. Sato, R. Takemura & M. Okada (2014): *Towards explaining the cognitive efficacy of Euler diagrams in syllogistic reasoning: A relational perspective*. *Journal of Visual Languages & Computing* 25(3), pp. 156–169, doi:10.1016/j.jvlc.2013.08.007.

[13] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman & M. Zenger (2004): *An Overview of the Scala Programming Language*. Technical Report IC/2004/64, EPFL.

[14] L. C. Paulson (1994): *Isabelle - A Generic Theorem Prover*. *LNCS* 828, Springer.

[15] A. Shimojima (2004): *Inferential and Expressive Capacities of Graphical Representations: Survey and Some Generalizations*. In: *Diagrammatic Representation and Inference*, *LNAI* 2980, Springer, pp. 18–21, doi:10.1007/978-3-540-25931-2_3.

[16] G. Stapleton, J. Masthoff, J. Flower, A. Fish & J. Southern (2007): *Automated Theorem Proving in Euler Diagrams Systems*. *Journal of Automated Reasoning* 39(4), pp. 431–470, doi:10.1007/s10817-007-9069-y.

[17] G. Stapleton, L. Zhang, J. Howse & P. Rodgers (2011): *Drawing Euler Diagrams with Circles: The Theory of Piercings*. *IEEE Transactions on Visualization and Computer Graphics* 17(7), pp. 1020–1032, doi:10.1109/TVCG.2010.119.

[18] M. Urbas & M. Jamnik (2012): *Diabelli: A Heterogeneous Proof System*. In B. Gramlich, D. Miller & U. Sattler, editors: *IJCAR*, *LNAI* 7364, Springer, pp. 559–566, doi:10.1007/978-3-642-31365-3_44.

[19] M. Urbas & M. Jamnik (2014): *A Framework for Heterogeneous Reasoning in Formal and Informal Domains*. In T. Dwyer, H.C. Purchase & A. Delaney, editors: *Diagrams*, *LNCS* 8578, Springer, pp. 277–292, doi:10.1007/978-3-662-44043-8_28.

[20] M. Urbas, M. Jamnik & G. Stapleton (2015): *Speedith: A Reasoner for Spider Diagrams*. *Journal of Logic, Language and Information*, pp. 1–54, doi:10.1007/s10849-015-9229-0.