Speedith: a diagrammatic reasoner for spider diagrams

Matej Urbas[1], Mateja Jamnik[1], Gem Stapleton[2], and Jean Flower[3]

[1] Computer Laboratory, University of Cambridge, UK
{Matej.Urbas,Mateja.Jamnik}@cl.cam.ac.uk
[2] School of Computing, Engineering and Mathematics, University of Brighton, UK
g.e.stapleton@brighton.ac.uk
[3] Autodesk, UK

Abstract. In this paper, we introduce Speedith which is a diagrammatic theorem prover for the language of spider diagrams. Spider diagrams are a well-known logic for which there is a sound and complete set of inference rules. Speedith provides a way to input diagrams, transform them via the diagrammatic inference rules, and prove diagrammatic theorems. It is designed as a program that plugs into existing general purpose theorem provers. This allows for seamless formal verification of diagrammatic proof steps within established proof assistants such as Isabelle. We describe the general structure of Speedith, the diagrammatic language, the automatic mechanism that draws the diagrams when inference rules are applied on them, and how formal diagrammatic proofs are constructed.

1 Introduction

Diagrams have been used to prove theorems since ancient times. One can argue that diagrams often provide compelling and intuitive solutions to problems. Despite this, diagrams have rarely been formalised in proof tools to be used for reasoning. In this paper, we do just that: we present a new, formal diagrammatic theorem prover Speedith. Speedith's domain is the language of spider diagrams. It allows us to apply diagrammatic inference rules on conjectures about spider diagrams, and thus construct a proof. The entire proof construction process is carried out visually. The derived proof is certified to be (logically) correct. Here are the hypotheses we test and objectives we aim to achieve:

– We want to show that it is possible to design and implement a complete formal diagrammatic reasoner in the general domain of monadic first-order logic (MFOL) with equality, expressed using the language of spider diagrams.
– We aim to have the guarantee that the derived proofs are formally correct.
– We aim for our system to be standalone, yet also reasonably easily plugable into external proof tools, thus providing alternative problem representation and proof construction method for these tools.

Whilst there exist other purely diagrammatic theorem provers, such as Diamond [1], Dr.Doodle [2], and Cinderella [3], they target different, more restricted domains (e.g., a small subset of natural number arithmetic, a subset of real arithmetic), and are hence able to prove only a limited class and number of theorems.

They do not provide a provably sound and complete set of inference rules. They are also not designed to be readily integrated into external proof tools.

There are theorem provers that were developed for spider diagrams, but they worked only for fragments of the logic in this paper, and did not include any logical connectives, or only a limited number of them [4]. In Speedith we formalize the whole spider diagram logic, which is expressively equivalent to MFOL with equality. We also develop a set of sound inference rules, representing a conservative extension of the complete system in [5], which allow for more intuitive proof steps.

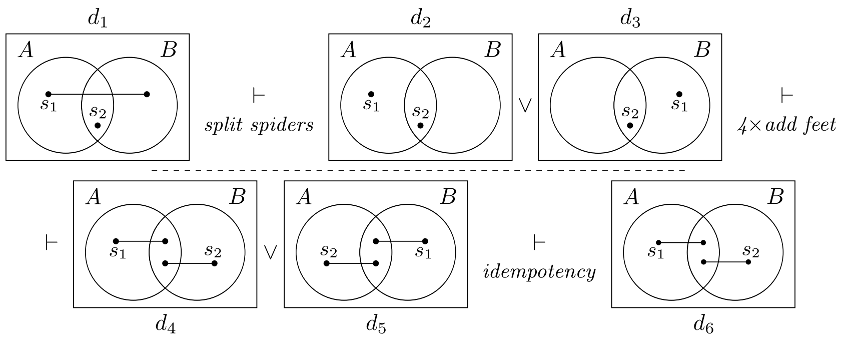

Fig. 1: A proof of a spider-diagrammatic statement. The proof establishes that given sets A and B, if there are two elements s_1 and s_2 and one is in both of A and B and the other is either in only A or only B, then we can deduce that one element is in A and the other is in B. In this proof, we applied the split spiders, add feet, and idempotency inference rules. The rules are proved to be sound and their application in this proof is verified by Speedith to be correct. Hence, the proof is certified to be correct.

Speedith is an interactive proof assistant for the language of spider diagrams and allows its users to interactively apply diagrammatic (visual) inference rules on spider-diagrammatic statements. It checks whether the inference rules are used correctly and verifies that a spider-diagrammatic statement expresses a true fact – it is a theorem. Thus, Speedith's diagrammatic proofs are entirely formal and certified to be correct. Fig. 1 shows an example of Speedith's purely diagrammatic proof. Here, d_1 is a spider diagram which conveys some information about the relationships between two elements and two sets and proves that d_6 follows logically.

Speedith provides a graphical user interface through which all the diagrammatic proofs are constructed. It visually displays spider-diagrammatic statements; allows the user to specify which inference rules should be applied on what parts of the spider diagram; and displays the result of this visually.

Whilst Speedith is a standalone diagrammatic proof assistant, it is also designed to easily plug into external proof tools. This has the advantage that spider-diagrammatic proofs can be reconstructed in traditional logic, and thus certified with, for example, LCF-style general purpose theorem provers [6].

To confirm our hypotheses above and achieve our aims, we designed Speedith as a standalone system that incorporates the following components: full specification of Spider diagrams (Sec. 2) and their inference rules (Sec. 3), a reasoning kernel that manages the state of the proofs, controls how inference steps are applied, and manages the communication with external general purpose theorem provers (Sec. 4), and a visualisation component with input methods for constructing diagrammatic statements and interactively applying inference rules (Sec. 5). Lastly, we evaluate our prover (Sec. 6) and conclude with future directions and general observations (Sec. 7).

2 Spider Diagrams: Syntax and Semantics

We now introduce spider diagrams (see [5] for more details and examples). Spider diagrams use closed curves, called *contours*, to represent sets and assert relationships between those sets. For instance, the enclosure of one contour by another contour corresponds to a subset/superset relationship between the represented sets. Contours are named with labels (in d_1 in Fig. 1, the contour labels are A and B). The set of contour labels used in a diagram d is denoted by $L(d)$.

A *zone* is a region in a diagram that is inside some of the contours (possibly no contours) and not inside the rest of them. Formally, a zone is a pair of finite, disjoint sets of contour labels, (in, out). Intuitively, (in, out) is inside every contour of *in*, and outside every contour of *out*. So, in a diagram, the set of possible zones is formed by its contour labels (e.g., in d_1, the zones are $(\emptyset, \{A, B\})$, $(\{A\}, \{B\})$, $(\{B\}, \{A\})$, $(\{A, B\}, \emptyset)$). We denote the set of zones in a diagram d by $Z(d)$. The zones from $Z(d)$ can be *shaded*, denoted $ShZ(d)$, and this places upper bounds on set cardinality: in a shaded zone, all elements are represented by spiders.

Spiders are trees used in spider diagrams to assert the existence of elements; they place lower bounds on set cardinalities (e.g., d_1 contains two spiders called s_1 and s_2 representing an element in only A or only B, and an element in A and B; there may be other elements too). The nodes of the trees are called *spider feet*, or simply *feet* (e.g., in d_1 the spider s_1 has two feet). The set of spiders in d is denoted by $S(d)$ and the function $\eta \colon S(d) \to \mathbb{P}Z(d) - \{\emptyset\}$ (here $\mathbb{P}$ denotes the power set) returns the set of zones in which the spider is placed, called its *habitat* (e.g., in Fig. 1, s_1 is placed in the zones $(\{A\},\{B\})$ and $(\{B\},\{A\})$). To avoid ambiguity when talking about the habitats of spiders in more than one diagram, we write $\eta_d(s)$ to mean the habitat of spider s in diagram d.

The diagrams considered so far are called *unitary* spider diagrams: they can be defined by a tuple $d = (L, Z, ShZ, S, \eta)$ as described above.[4] Spider diagrams can be negated and joined with binary connectives into *compound* diagrams: $\neg$ to denote 'not', $\wedge$ to denote 'and', $\vee$ to denote 'or', $\Rightarrow$ to denote 'implies' (e.g., in Fig. 1, $d_2 \vee d_3$ forms a compound diagram), and $\Leftrightarrow$ to denote 'equivalent'.[5]

[4] We define unitary diagrams differently to [5] where no mapping function for spiders and their habitats was used. The mapping function provides a convenient translation of spiders into formulae where each spider is a variable (see Sec. 4.3).

[5] This extends the definition of compound diagrams in [5] which did not allow $\neg$ or $\Rightarrow$.

The semantics of spider diagrams are captured by *interpretations*, $I = (U, \Psi)$, where U is a universal set, and Ψ is an assignment of a subset of U to each contour label. A zone, (in, out), represents the set:

$$\Psi(in, out) = \bigcap_{l \in in} \Psi(l) \cap \bigcap_{l \in out} (U - \Psi(l))$$

where $\Psi(l)$ is the set assigned to contour label l. A set of zones represents the set which is the union of the sets represented by the individual zones.

In order to identify when an interpretation agrees with the meaning of a unitary diagram d we define *missing zones*, $MZ(d)$ such that:

$$MZ(d) = \{(in, L(d) - in) : in \subseteq L(d) \wedge (in, L(d) - in) \notin Z(d)\}.$$

Intuitively, the missing zones are the zones that do not appear in the diagram due to its particular configuration, but could be specified using the labels from $L(d)$ (e.g., consider d_1 in Fig. 2 on page 5; nine zones can be specified, but do not appear in d_1, including $(\{A, C, D\}, \{B\})$ and $(\{A, B\}, \{C, D\})$). Briefly, we say that an interpretation, $I = (U, \Psi)$, is a *model* for unitary diagram d if there exists a function $\psi\colon S(d) \to U$ (interpreting the spiders as elements) that ensures:

1. the missing zones represent the empty set, that is, $\Psi(MZ(d)) = \emptyset$;
2. each spider s in d maps to an element $\psi(s)$ of the set represented by the spider's habitat, that is, $\psi(s) \in \Psi(\eta(s))$;
3. no two spiders map to the same element, that is, $\psi(s_1) = \psi(s_2) \Rightarrow s_1 = s_2$;
4. the shaded zones contain only elements represented by spiders, that is, $\Psi(ShZ(d)) \subseteq \{\psi(s) : s \in S(d)\}$.

The definition of a model extends in the obvious way to compound diagrams.

3 Speedith's Inference Rules

We present some of Speedith's inference rules for spider diagrams. We introduce three new rules that conservatively extend the set of sound and complete rules from [5]. Speedith can use all of them, but we only present the ones needed for our examples. Our new rules are designed to allow making intuitive proof steps and to substantially reduce proof length. All inference rules are proved to be sound but, due to space restrictions, we omit the proofs in this paper.

Our first rule allows us to 'copy' a contour from one diagram into another diagram: we argue that this is a natural deduction step. To illustrate, consider the diagram $d_1 \wedge d_2$ in Fig. 2. In d_2 we can see that $C \subseteq E$ and $E \subseteq A$. We can copy the contour E from d_2 to d_1 using this information and thus obtain d_1'. Here, E is completely inside A (since $E \subseteq A$) and C is completely inside E (since $C \subseteq E$). We do not know anything about the relationship between E and D, thus we ensure that E partially overlaps with D. The spider habitats are updated in line with how the zones have changed. In general, all shading is preserved in the same fashion.

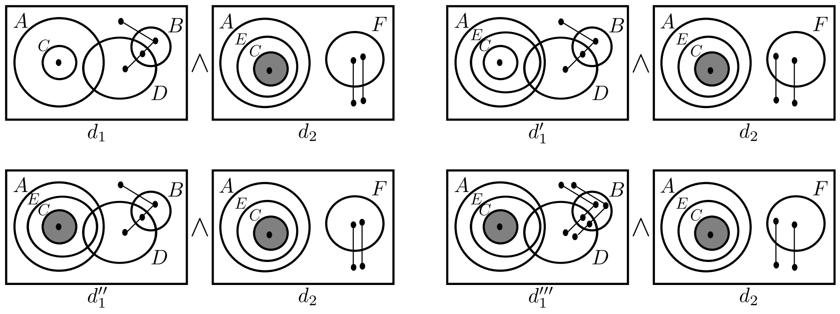

Fig. 2: Illustrating new inference rules: copying syntax.

To define the copy contour rule, we start by observing that each zone in the diagram, d_1, into which the contour is copied is either (a) completely outside the new contour, (b) completely inside the new contour, or (c) split into two zones by the new contour. In order to identify what happens to each zone, we need to inspect the contours of d_2.

Definition 1. *Let d be a unitary diagram and let λ and λ' be in $L(d)$.*

1. *If (the contours labelled) λ and λ' have disjoint interiors then λ and λ' are **disjoint** in d, denoted $\lambda \cap_d \lambda' = \emptyset$.*
2. *If λ is in the interior of λ' then λ is a **contained** by λ' in d, denoted $\lambda \subseteq_d \lambda'$.*

For example, in Fig. 2, inspecting d_2 we have the following relations involving E: $E \cap_{d_2} F = \emptyset$, $E \subseteq_{d_2} A$ and $C \subseteq_{d_2} E$. Using these relations, we can determine the effect of copying E from d_2 to d_1 on the zones. In particular, since $E \subseteq_{d_2} A$, all zones that are not inside A will not be inside E; these zones are placed in a set called Z_{OUT} (OUT for 'outside').[6] Since $C \subseteq_{d_2} E$, all zones that are inside C will be inside E; these zones are placed in a set called Z_{IN} (IN for 'inside'). The remaining zones will be split when E is copied into d_1.

Definition 2. *Let d_1 and d_2 be unitary diagrams and let λ be in $L(d_2) - L(d_1)$. We define three subsets of $Z(d_1)$ ($Z_{OUT}(\lambda, d_2)$, $Z_{IN}(\lambda, d_2)$, and $Z_{SPLIT}(\lambda, d_2)$) according to the following rules: let $(in, out) \in Z(d_1)$*

1. *$(in, out) \in Z_{OUT}(\lambda, d_2)$ provided there exists a contour label, λ', in $L(d_2)$ such that either*
   (a) $\lambda' \in in$ and $\lambda \cap_{d_2} \lambda' = \emptyset$, or
   (b) $\lambda' \in out$ and $\lambda \subseteq_{d_2} \lambda'$,
2. *$(in, out) \in Z_{IN}(\lambda, d_2)$ provided there exists a contour label, λ', in $L(d_2)$ such that $\lambda' \in in$ and $\lambda' \subseteq_{d_2} \lambda$,*
3. *finally, $Z_{SPLIT}(\lambda, d_2) = Z(d_1) - (Z_{IN}(\lambda, d_2) \cup Z_{OUT}(\lambda, d_2))$.*

Rule 1 ***Copy a Contour*** *Let d_1 and d_2 be unitary diagrams and let λ be in $L(d_2) - L(d_1)$. Let d_1' be the diagram whose components are defined as follows:*

[6] If F occurred in d_1 then, since $E \cap_{d_2} F = \emptyset$ all zones inside F would also be outside E.

1. the contour labels are $L(d_1') = L(d_1) \cup \{\lambda\}$,
2. the zones are

$$Z(d_1') = \{(in, out \cup \{\lambda\}) : (in, out) \in Z_{OUT}(\lambda, d_2) \cup Z_{SPLIT}(\lambda, d_2)\} \cup \{(in \cup \{\lambda\}, out) : (in, out) \in Z_{IN}(\lambda, d_2) \cup Z_{SPLIT}(\lambda, d_2)\}$$

3. the shaded zones are

$$ShZ(d_1') = \{(in, out \cup \{\lambda\}) : (in, out) \in (Z_{OUT}(\lambda, d_2) \cup Z_{SPLIT}(\lambda, d_2)) \cap ShZ(d_1)\} \cup \{(in \cup \{\lambda\}, out) : (in, out) \in (Z_{IN}(\lambda, d_2) \cup Z_{SPLIT}(\lambda, d_2)) \cap ShZ(d_1)\}$$

4. the spiders are $S(d_1') = S(d_1)$, and
5. the habitat of each spider, $s' \in S(d_1')$, is

$$\eta_{d_1'}(s') = \{(in, out \cup \{\lambda\}) : (in, out) \in (Z_{OUT}(\lambda, d_2) \cup Z_{SPLIT}(\lambda, d_2)) \cap \eta_{d_1}(s')\} \cup \{(in \cup \{\lambda\}, out) : (in, out) \in (Z_{IN}(\lambda, d_2) \cup Z_{SPLIT}(\lambda, d_2)) \cap \eta_{d_1}(s')\}.$$

Then $d_1 \wedge d_2$ is logically equivalent to $d_1' \wedge d_2$.

As well as enabling 'natural' proof steps to be made in a single inference step, our copy contour substantially reduces the number of proof steps required. In Fig. 2, if we used only the inference rules from [5], a proof establishing $d_1 \wedge d_2 \vdash d_1' \wedge d_2$ would require hundreds of proof steps (in part because, using the inference rules of [5], in proofs that $d_1 \wedge d_2 \vdash d_1' \wedge d_2$ the number of spider feet increases rapidly when contours are added and all of these spiders need to be split until they have single feet).

If we consider $d_1' \wedge d_2$ in Fig. 2, we can see that the shaded region that comprises the zone $(\{A, E, C\}, \{F\})$ in d_2 represents the set C (this is the only zone inside C). There is a corresponding zone in d_1', namely $(\{A, C\}, \{B, D\})$, that also represents the set C. Since these two zones contain the same spiders, we can copy the shading from d_2 over to d_1', as shown in $d_1'' \wedge d_2$. In order to define this rule, we introduce some notation to denote the set of spiders whose habitat is a subset of a given region r in a unitary diagram d: $S(r, d) = \{s \in S(d) : \eta(s) \subseteq r\}$. In addition, it is possible to syntactically identify when two distinct regions, r_1 and r_2, necessarily represent the same set [7]. Such regions are said to be *corresponding*. To illustrate the idea, in Fig. 2 the region inside d_1 that comprises the four zones outside of A corresponds to the region that comprises the two zones outside of A in d_2.

Rule 2 ***Copy Shading*** *Let d_1 and d_2 be unitary diagrams with corresponding regions, r_1 and r_2 respectively, such that:*

1. r_1 contains at least one non-shaded zone in d_1,
2. r_2 is entirely shaded in d_2,
3. in d_1, all of the spiders that have a foot in r_1 are also in $S(r_1, d_1)$,
4. in d_2, all of the spiders that have a foot in r_2 are also in $S(r_2, d_2)$, and
5. there is a habitat preserving bijection, σ, from $S(r_1, d_1)$ to $S(r_2, d_2)$ (i.e., $\eta_{d_1}(s)$ corresponds to $\eta_{d_2}(\sigma(s))$).

Let d'_1 be a copy of d_1 except that r_1 is entirely shaded. Then $d_1 \wedge d_2$ is logically equivalent to $d'_1 \wedge d_2$.

Our final new rule allows us to copy spiders from one diagram to another. This is illustrated in Fig. 2, where we can copy a spider from d_2 into d''_1, to give $d'''_1 \wedge d_2$. The two spiders in d_2 that inhabit the region outside of A tell us that there are at least two elements in $U - A$, where U is the universal set. Since there is only one spider in the corresponding region of d''_1, that is, there is at least one element in $U - A$, we can copy across the second spider.

Rule 3 ***Copy a Spider*** *Let d_1 and d_2 be unitary diagrams with corresponding regions, r_1 and r_2 respectively, such that:*

1. *r_1 contains no shaded zones in d_1,*
2. *in d_1, all of the spiders that have a foot in r_1 are also in $S(r_1, d_1)$,*
3. *there exists a habitat preserving injective, but not surjective, map σ from $S(r_1, d_1)$ to $S(r_2, d_2)$ (i.e., $\eta_{d_1}(s_1)$ corresponds to $\eta_{d_2}(\sigma_r(s_1))$).*

Choose a spider, s, that is in $S(r_2, d_2)$ but is not in the image of σ such that there exists a region, r' in d_1 that corresponds to $\eta_{d_2}(s)$. Let d'_1 be a copy of d_1 except d'_1 contains s with habitat r'. Then $d_1 \wedge d_2$ is logically equivalent to $d'_1 \wedge d_2$.

Three other rules are used in our examples: *add feet* and *split spiders* (see both in Fig. 1) and *remove a contour* (see Fig. 7); their formal definitions are in [5].

Rule 4 ***Add feet to a Spider*** *Let d_1 be a unitary diagram that contains a spider, s, whose habitat does not include all of the zones in d_1. Let d_2 be a copy of d_1 except that s contains additional feet. Then d_1 logically entails d_2.*

Rule 5 ***Split Spiders*** *Let d be a unitary diagram containing a spider, s, with habitat such that $|\eta(s)| \geq 2$. Let η_1 and η_2 be a two-way partition of $\eta(s)$. Let d_1 (d_2) be the diagram obtained from d by changing the habitat of s to η_1 (η_2). Then d and $d_1 \vee d_2$ are logically equivalent.*

Rule 6 ***Remove a Contour*** *Let d_1 be a unitary diagram and let $\lambda \in L(d)$. Let d_2 be the diagram obtained from d_1 by removing the contour, C, labelled λ, so $L(d_2) = L(d_1) - \{\lambda\}$. If, on the removal of C, two zones combine to form a single zone then spiders' habitats are updated in the same way. With regard to shading, if a shaded zone merges with a non-shaded zone then the shading is removed. Otherwise the shading remains. Then d_1 logically entails d_2.*

4 Architecture of Speedith

Speedith is the implementation of a diagrammatic theorem prover for spider diagrams described in Sec. 2, and the inference rules from Sec. 3 and [5]. It consists of four main components:

1. abstract representation of spider-diagrammatic statements (Sec. 4.1),
2. the reasoning kernel with proof infrastructure (Sec. 4.2),
3. verification of diagrammatic proofs, including input and output system for importing and exporting formulae in many different formats (Sec. 4.3), and
4. visualisation of spider-diagrammatic statements (Sec. 5).

4.1 Abstract representation

Speedith uses an abstract spider diagram representation to express spider-diagrammatic formulae. This representation is captured by the class diagram in Fig. 3. The null spider diagram is the unitary diagram containing only the zone $(\emptyset, \emptyset)$, no spiders and no shading; it is used as the logical truth constant $\top$. Unitary spider diagrams contain the bulk of diagrammatic information. Finally, the compound spider diagrams build up more complex formulae by connecting spider diagrams through the usual logical connectives: conjunction, disjunction, implication, equivalence and negation. Thus, a compound spider diagram nests one or more other spider diagrams, as indicated with the diamond notation in Fig. 3.

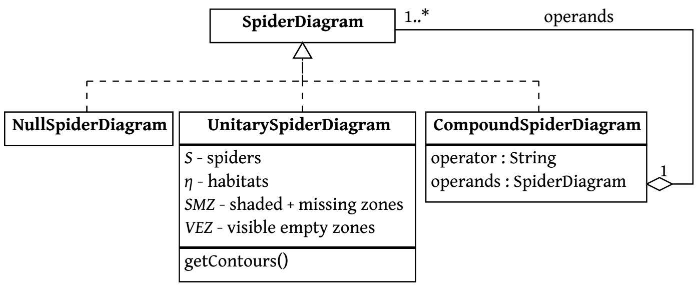

Fig. 3: A class diagram of the abstract representation of spider diagrams in Speedith.

To optimize performance, Speedith's abstract representation of spider diagrams removes some redundancies from the syntax in Sec. 2. In particular, Speedith does not explicitly store the sets $L(d)$ and $Z(d)$. Moreover, the sets ShZ and MZ are merged into SMZ, and the set VEZ lists all the zones that are shaded but have no spider feet in them. This more closely matches the semantics of spider diagrams as the zones that convey no semantic information (i.e. zones with no spider feet and no shading) are not explicitly stored. However, Speedith does store the shaded zones, the missing zones, and the spiders with their habitats, which is needed when converting diagrams to sentential form for verification. Note that all sets from the tuple $d = (L, Z, ShZ, S, \eta)$ in Sec. 2, can still be computed. The set $L(d)$ is obtained via the method `getContours()`, which takes an arbitrary zone (in, out) and computes $L(d) = in \cup out$. The set $MZ(d)$ is obtained through $\mathit{SMZ} - (\mathit{VEZ} \cup \text{habitats})$. Finally, $Z(d) = \{(in, L(d) - in) : in \subseteq L(d)\} - MZ(d)$.

An advantage of Speedith's abstract representation is the simplicity of converting spider-diagrammatic formulae to and from first-order logic formulae for the purposes of verification (see Sec.4.3). On the other hand, the user has no control on how the diagrams are drawn – Speedith lays them out automatically.

4.2 The reasoning kernel

One of the central components of Speedith is the reasoning kernel. It is responsible for correctly applying the inference rules on particular parts of spider-diagrammatic formulae. The kernel contains two types of inference rules: the ones for logical connectives based on the well-known logical equivalences, and purely spider-diagrammatic rules including the ones introduced in Sec. 3.

4.2.1 Proofs in Speedith A proof in Speedith starts with a spider-diagrammatic formula D (the *initial goal*, i.e., the theorem we aim to prove), proceeds by transforming D with applications of inference rules, and ends with an empty set of goals. The specific rules and parts of the diagram on which they should be applied are chosen by the user. Rules can be applied in both backward and forward reasoning styles. Backward proof steps take a goal D and transform it into a new goal D', where $D' \vdash D$. In forward proofs, D must be of the form $D_i \Rightarrow D_j$, and the rules transform D_i into D_i', where $D_i \vdash D_i'$, resulting in the new goal $D_i' \Rightarrow D_j$.[7] The user can switch between backward and forward proof styles due to the inference rule $(\top \Rightarrow D) \vdash D$. Either way, since the inference rules in both proof styles adhere to the entailment property for valid deductive steps, the proofs are of the standard structure:

$$\begin{array}{cl} \top & \\ \hline & \text{SD inference rule} \\ \vdots & \\ \hline & \text{SD inference rule} \\ \text{D'} & \\ \hline & \text{SD inference rule} \\ \text{D} & \end{array} \tag{1}$$

which together with the soundness of our rules justifies that D is a theorem.

Speedith stores and manages the proof as a sequence of goals and inference rule applications. The diagram in Fig. 4 outlines the architecture for managing proof state and goals. Multiple simultaneous goals are supported, and the proof is finished only when all goals are converted to null diagrams.

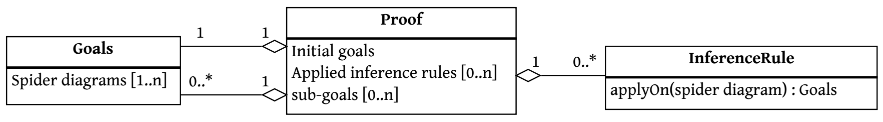

Fig. 4: A simplified class diagram of the part of the reasoning kernel responsible for tracking the proof state.

Once a spider-diagrammatic theorem D_t is proved, it is added to the database of theorems available for reuse in other proofs. This is possible in both backward

[7] Note that all our examples in Figs. 1, 6 and 7 use forward reasoning style.

and forward proofs through the *schematic inference rules* $\top \vdash D_t$ and $D_t \vdash \top$ respectively. In addition to theorems, Speedith allows the use of *axioms*: given an axiom D_a, it can be used in a proof through the inference rule $\top \vdash D_a$.

4.2.2 Transforming spider diagrams A compound spider diagram connects multiple sub-diagrams with logical connectives. In the abstract representation, sub-diagrams are children nodes of a compound spider diagram. This composition forms a tree structure. Every sub-diagram is assigned a sequential number – node indexes. The depth-first algorithm is used for assignment of these indexes. An example of the tree structure of a compound spider diagrams is shown in Fig. 5.

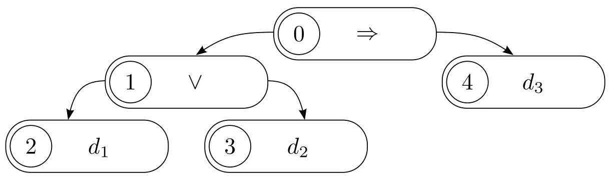

Fig. 5: The tree structure of the spider-diagrammatic formula $(d_1 \vee d_2) \Rightarrow d_3$, where d_1, d_2, and d_3 refer to unitary spider diagrams, and the logical connectives $\Rightarrow$ and $\vee$ refer to compound spider diagrams. The circled numbers are the indexes of sub-diagrams.

When an inference rule is to be applied, the tree structure of a spider diagram is traversed to a particular point where the relevant transformation takes place and thus returns new spider diagrams. This point is chosen by the user and supplied to the inference rule through the *rule application arguments*. The rule application arguments differ from rule to rule. Some rules, like the split spider rule, work on particular spiders in a unitary diagram. Thus, the split spider rule requires the sub-diagram index of the unitary diagram, the name of the spider, and also the region on which to split the spider's habitat. If a rule cannot be applied at a certain position, or if the sub-diagram does not satisfy all the requirements needed for a safe and valid rule application, the rule will not be applied and the user will be notified of this.

In the implementation of Speedith, the actual data structures of the abstract representation of a spider diagram do not change during the rule application. Rather, an entirely new spider diagram is constructed. For lower memory consumption and for efficiency purposes, the new spider diagram shares all unchanged sub-diagrams of the initial formula. This also improves the speed of syntactic equality comparisons in Speedith, as there cannot exist two different instances of syntactically identical spider diagrams.

4.3 Verification with external tools

Proofs in Speedith rely purely on the soundness of the individual inference rules outlined in Sec. 3 and [5]. As proofs are derived by sequential application of these rules, they are guaranteed to be correct by construction (see Sec. 4.2.1).

However, the implementation of Speedith cannot be guaranteed to be bug-free. Also, some users might trust Speedith more if its proofs are verified in other, established, theorem provers. Thus, we designed Speedith to easily plug into, and allow for seamless communication with external proof tools. This enables Speedith to verify particular proof steps externally: namely, spider-diagrammatic proofs can be reconstructed in traditional logic. To date, Speedith's proofs can be certified in the general purpose theorem prover Isabelle [8]. Other tools can be supported by supplying a plug-in which implements the communication with the new external tool. Conceptually, a proof step in Speedith can be proved correct by verifying that conjunctively connected sub-goals D_i imply the initial goal D (see Formula (1) in Sec. 4.2.1). Thus, in order to verify its proof steps, Speedith exports the following theorem which is to be proved by the external tool:

$$D_1 \wedge D_2 \wedge \cdots \wedge D_n \Rightarrow D.$$

Apart from exporting, Speedith can also import formulae. This enables the so-called heterogeneous reasoning, that is, constructing proofs that consist of diagrammatic inferences and traditional sentential logical inferences. We use it in our heterogeneous reasoning framework called Diabelli [8] that combines diagrammatic theorem proving in Speedith with sentential theorem proving in Isabelle [9].

4.3.1 Input and output formats Speedith supports different input and output formulae formats for importing and exporting. The standard input format of Speedith is the native textual representation outlined in Sec. 4.3.2 below. For export, on the other hand, Speedith uses formats that several other tools understand. For example, one supported export format translates abstract representations of spider diagrams into Isabelle/HOL formulae. Here is an example of how the diagram d_1 in Fig. 1 is translated to the Isabelle/HOL format:

```
∃s1 s2. distinct[s1, s2] ∧ s1 ∈ A ∩ B ∧ s2 ∈ (A - B) ∪ (B - A)
```

A new output or input format can be specified by providing a translation procedure that takes a set of spider diagrams in their abstract representation, a proof trace, or an inference rule application, and translates it to a specific textual format.

4.3.2 Textual representation In addition to the data structures and object oriented model used in abstract representation, Speedith also provides a textual form of spider diagrams. This form is Speedith's default for exporting and importing diagrammatic statements to and from external tools. It is used when verifying particular steps in the diagrammatic proof, or verifying properties about spider-diagrammatic formulae in a sentential reasoner. Here is an example of the textual form of diagram d_1 in Fig. 1:

```
PrimarySD {
  spiders  = ["s1", "s2"],
  habitats = [("s1", [(["A"], ["B"]), (["B"], ["A"])]),
              ("s2", [(["A", "B"], [])])],
  sm_zones = [], ve_zones = []
}
```

5 Diagram Visualisation

The visualization component of Speedith, called iCircles, builds on the Euler diagram drawing software presented in [10]. We extended the drawing algorithm to spider diagrams, enabling the layout of spiders as well as including functionality to specify which zones are to be shaded.

The input to Speedith is a statement of the theorem. This can be entered through drawing commands, first-order logic formulae,[8] or the textual representation (as described in Sec. 4.3.2). A future direction would be to support additional input methods, such as free-hand drawing and shape recognition.

Next, the entered theorem is automatically drawn – Speedith uses the following drawing algorithm:

1. Draw the underlying Euler diagram, using the methods of [10]. This takes the set of zones to be present and determines how to draw the diagram with circles.
2. Next, shading is placed in the appropriate zones, using standard 'region shading' methods.
3. Finally, the spiders are laid out. Given a particular spider, s, with habitat $\eta(s)$, a point is found in each zone in $\eta(s)$. These points form the feet of the spider. The feet are then joined by edges: we prioritize drawing edges between feet in adjacent zones, until the feet form a tree. Care is taken to ensure that edges do not pass through feet belonging to other spiders. This is achieved by nudging the position of feet that lie on the path of a to-be-drawn edge in eight principal directions until a suitable position is found. In addition, when the initial points are found for spiders, we ensure that they do not lie on already drawn edges.

Fig. 6 is an example of Speedith using iCircles to display spider-diagrammatic formulae. The abstract representation of a spider-diagrammatic formula is converted into iCircles and composed to form arbitrary compound spider diagrams.

Speedith translates the abstract representation of unitary spider diagrams into a concrete description of the drawing. An important step of the translation is determining which shaded zones contain spider feet and must thus be present in the drawing. The decision on whether to display other shaded zones is left to the iCircles drawing algorithm.

After all unitary spider diagrams of a compound statement are drawn, they are laid out as operands of the logical connectives. Individual sub-diagrams are finally enclosed with a bounding box, which separates them spatially for clearer presentation and unambiguous nesting of operators.

Following the entry and drawing of the theorem, the user proceeds to apply inference rules on specific parts of the diagram. The exact target of the rule application is determined through a rule-specific sequence of clicking on select elements in the diagram. The inference rule to apply is chosen from a list of

[8] Speedith currently supports only a specific form of first-order logic formulae in the Isabelle/HOL syntax.

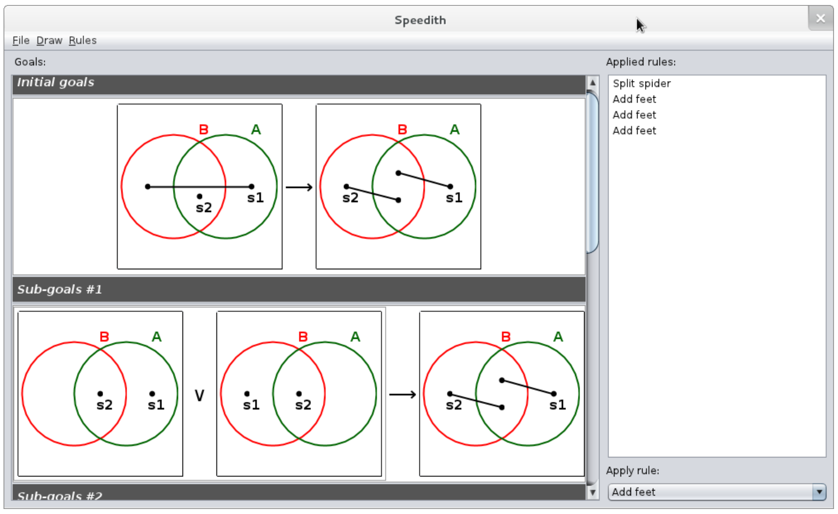

Fig. 6: Automatically drawing spider diagrams using iCircles.

all available rules; as we implement more inference rules in the future, we will introduce a filter that shows only the rules applicable to the part of the diagram that the user clicked on. Once the proof is completed, it is added to the suite of unit tests.

6 Results and Related Work

Speedith is implemented in Java and is currently under active development. Its sources are available from `https://gitorious.org/speedith`. With Speedith we are able to prove all theorems of MFOL with equality, expressed using spider diagrams – this is a significant range and depth of theorems.

We demonstrate the evaluation of Speedith's functionality with two diagrammatic proof examples. The first one was presented in Fig. 1. The proof makes use of diagrammatic rules that transform spiders and a disjunction equivalence rule. The second example, shown in Fig. 7, tests the inference rules which manipulate spiders, contours, shaded zones, and logical connectives. The proof in Fig. 7 essentially makes use of the information from the right conjunct to transform the diagram representing the left conjunct through a series of copying rules: first the contour D, then spider s_1, followed by shading of the zone $(\{C\}, \{D\})$, and finally eliminating the redundant right conjunct and removing contours A and C to deduce the theorem's conclusion.

One of Speedith's main contributions is its representation of formulae and proof steps. This differentiates it from interactive sentential theorem provers (such as Isabelle) in that it provides a domain-specific, visual, and thus perhaps more intuitive approach to proofs in MFOL with equality. Speedith's inference rules, which perform simple visual transformations of the diagrammatic statement are succinct and 'natural' – they capture the notion of truthfulness that

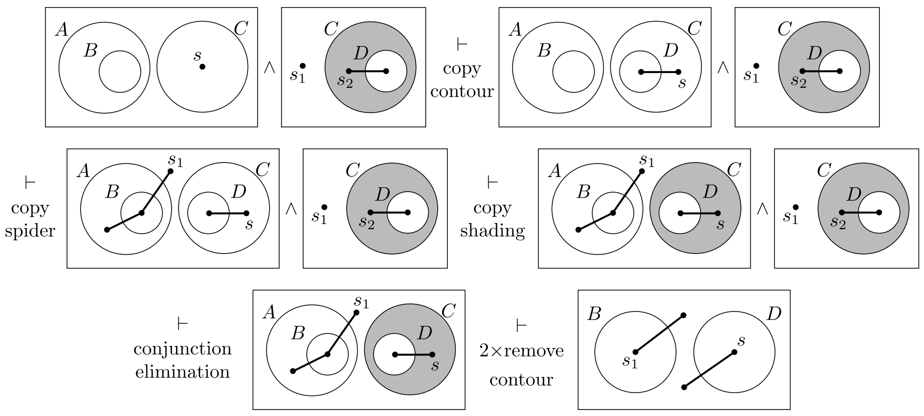

Fig. 7: A proof of a spider-diagrammatic statement using inference rules that work with spiders, contours, and shaded zones.

humans find easy to understand. In contrast, proofs of the same theorems in sentential theorem provers consist of lower-level, more fine-grained proof steps which make them longer and arguably harder to "see" the intuition behind the proof. Comparing Speedith's speed with other theorem proving tools remains work for the future.

Other diagrammatic theorem provers most related to Speedith are Edith, DIAMOND, and Cinderella. Whilst Edith is the closest to Speedith in terms of the domain it targets, it does not support spiders nor compound diagrams with logical connectives, and thus provides fewer inference rules. Edith also does not support external verification of its proof steps. DIAMOND, on the other hand, supports external verification, but the class of problems it tackles is different and narrower compared to Speedith. Cinderella targets the domain of geometry and uses a different approach to its diagrammatic proofs. The user gradually constructs the geometric model of the theorem, while in the background an automated theorem prover verifies that each construction step results in a valid geometric diagram. Thus, the steps in Cinderella are not guaranteed to be sound, and the proof process does not follow the standard inference rule application pattern (as described in Sec. 4.2.1).

Finally, Speedith was designed with language extensions in mind. Spider diagrams could be extended with non-monadic relations, functions, and universal quantification of spiders. Designing meaningful and complete diagrammatic inference rules for such extended language is hard and remains work for the future.

7 Future Work and Conclusion

By developing Speedith, we demonstrated the feasibility of diagrammatic reasoning systems that utilise a rule-based deductive proof approach. This is similar to the approach employed by general purpose proof assistants like Isabelle.

We also showed how to utilize existing state-of-the-art theorem provers to verify diagrammatic inference steps. Whilst we focused on spider diagrams, the approach can be used for other diagrammatic logics, such as existential graphs [11] or constraint diagrams [12].

Part of our future directions for Speedith includes extending the abstract representation to better control how diagrams are drawn. Moreover, the diagrams are currently laid out independently, and hence diagrams in consecutive proof steps can look radically different from each other. Thus, we aim to improve layout heuristics to take entire sequences of diagrammatic statements into account. In addition to better diagram visualisation, we also envision extensions to the language of spider diagrams, proof search automation, use of Speedith in practical settings [13,14], and a study of scalability of proofs and their visualisation in Speedith.

Acknowledgements This work was supported by EPSRC Advanced Research Fellowship GR/R76783 (Mateja Jamnik), EPSRC Doctoral Training Grant and Computer Laboratory Premium Research Studentship (Matej Urbas).

References

1. M. Jamnik, A. Bundy, and I. Green. On Automating Diagrammatic Proofs of Arithmetic Arguments. *JOLLI*, 8(3):297–321, 1999.
2. D. Winterstein, A. Bundy, and C.A. Gurr. Dr.Doodle: A Diagrammatic Theorem Prover. In *IJCAR*, volume 3097 of *LNAI*, pages 331–335, 2004.
3. U. Kortenkamp and J. Richter-Gebert. Using automatic theorem proving to improve the usability of geometry software. In *MUI*, 2004.
4. G. Stapleton, J. Masthoff, J. Flower, A. Fish, and J. Southern. Automated Theorem Proving in Euler Diagram Systems. *JAR*, 39(4):431–470, 2007.
5. J. Howse, G. Stapleton, and J. Taylor. Spider Diagrams. *LMS JCM*, 8:145–194, 2005.
6. M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.
7. J. Howse, G. Stapleton, J. Flower, and J. Taylor. Corresponding Regions in Euler Diagrams. In *Diagrams*, volume 2317 of *LNCS*, pages 76–90. Springer, 2002.
8. M. Urbas and M. Jamnik. Heterogeneous proofs: Spider diagrams meet higher-order provers. In *ITP*, volume 6898 of *LNCS*, pages 376–382. Springer, 2011.
9. M. Wenzel, L. C. Paulson, and T. Nipkow. The Isabelle Framework. In *TPHOLs*, volume 5170 of *LNCS*, pages 33–38. Springer, 2008.
10. G. Stapleton, J. Flower, P. Rodgers, and J. Howse. Drawing Euler Diagrams with Circles. In *Diagrams*, volume 6170 of *LNCS*, pages 23–38. Springer, 2010.
11. F. Dau. Constants and functions in peirce's existential graphs. In *ICCS*, volume 4604 of *LNCS*, pages 429–442. Springer, 2007.
12. S. Kent. Constraint diagrams: Visualizing invariants in object oriented modelling. In *OOPSLA*, volume 32 of *SIGPLAN*, pages 327–341. ACM, 1997.
13. H. Keslter, A. Muller, J. Kraus, M. Buchholz, T. Gress, H. Liu abd D. Kane, B. Zeeberg, and J. Weinstein. Vennmaster: Area-proportional Euler diagrams for functional go analysis of microarrays. *BMC Bioinformatics*, 9(67), 2008.
14. R. De Chiara, M. Hammar, and V. Scarano. A system for virtual directories using euler diagrams. *ENTCS*, 134:33–53, 2005.