Taming Differentiable Logics with Coq Formalisation

Reynald Affeldt
National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan

Alessandro Bruni
IT-University of Copenhagen, Denmark

Ekaterina Komendantskaya
Southampton University, UK
Heriot-Watt University, Edinburgh, UK

Natalia Ślusarz
Heriot-Watt University, Edinburgh, UK

Kathrin Stark
Heriot-Watt University, Edinburgh, UK

Abstract

For performance and verification in machine learning, new methods have recently been proposed that optimise learning systems to satisfy formally expressed logical properties. Among these methods, differentiable logics (DLs) are used to translate propositional or first-order formulae into loss functions deployed for optimisation in machine learning. At the same time, recent attempts to give programming language support for verification of neural networks showed that DLs can be used to compile verification properties to machine-learning backends. This situation is calling for stronger guarantees about the soundness of such compilers, the soundness and compositionality of DLs, and the differentiability and performance of the resulting loss functions. In this paper, we propose an approach to formalise existing DLs using the Mathematical Components library in the Coq proof assistant. Thanks to this formalisation, we are able to give uniform semantics to otherwise disparate DLs, give formal proofs to existing informal arguments, find errors in previous work, and provide formal proofs to missing conjectured properties. This work is meant as a stepping stone for the development of programming language support for verification of machine learning.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Logic and verification; Computing methodologies → Rule learning

Keywords and phrases Machine Learning, Loss Functions, Differentiable Logics, Logic and Semantics, Interactive Theorem Proving

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.4

Supplementary Material *Software (Source)*: `https://github.com/ndslusarz/formal_LDL` [29] archived at `swh:1:dir:bd213b761dfc453ccfe8e785a38cffe583c98f04`

1 Introduction

This work aims to contribute to the field of formal verification of artificial intelligence, more precisely machine learning, i.e., the study of algorithms that learn statistically from data. Neural networks are the most common technical device used in machine learning. The standard learning algorithms (such as gradient descent) use a *loss function* $\mathcal{L} : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$ to optimise the network's parameters (say, θ) to fit the input-output vectors given by the data in a way that the loss $\mathcal{L}(\mathbf{x}, \mathbf{y})$ is minimised. This optimisation objective is usually denoted as $\min_\theta \mathcal{L}(\mathbf{x}, \mathbf{y})$.


© Reynald Affeldt, Alessandro Bruni, Ekaterina Komendantskaya, Natalia Ślusarz, and Kathrin Stark; licensed under Creative Commons License CC-BY 4.0

Most approaches to verification of neural networks consist of an automated procedure based on SMT solving, abstract interpretation, or branch-and-bound techniques (see, e.g., Albarghouthi's survey [3]). Verification typically applies after training because traditional learning is purely data-driven and thus agnostic to verification properties. In contrast, *property-guided training* takes place once the verification properties are stated. More precisely, verification of neural networks consists of two parts: statement and verification of a given property, and training of the neural network that optimises the neural network's parameters towards satisfying the given property.

However, naively or manually performed mapping of a logical property to an optimisation task results in major discrepancies (as shown by Casadio et al. [9]). This suggests the need to have tools for property-guided training. One approach is to provide programming language support for property-driven development of neural networks that involves specification, verification, and optimisation in a safe-by-construction environment. As an example, Vehicle [11, 13] provides this support in the form of a Haskell DSL, with a higher-order typed specification language, in which required neural network properties can be clearly documented, and type-driven compilation which can take care of correct-by-construction translation of properties into both the language of neural network solvers and loss functions.

To generate loss functions from a logical property, one can use *Differentiable Logics* (DLs). Well-studied *fuzzy logics* that date back to the works of Łukasiewicz and Gödel can be used as DLs [26]. Recently, both verification and machine-learning communities formulated alternative DLs such as DL2 [16] and STL [27]; the latter was shown to be more performant in optimisation tasks. These DLs are very different; for example, they do not agree on the domains of the resulting loss functions: fuzzy logics have domain $[0, 1]$, the domain of DL2 corresponds to the Lawvere quantale $(-\infty, 0]$, and STL's domain is $(-\infty, +\infty)$ (all intervals are equipped with the usual ordering on reals). Each domain has a designated value for truth (e.g., 1 in fuzzy logics, 0 in DL2, $+\infty$ in STL) and falsity (0, $-\infty$, $-\infty$, respectively).

Vehicle uses DLs to translate logical properties into loss functions. In order to ensure the correctness of the translation, a DL needs to satisfy a number of properties:

- *Soundness*: if a property interprets as "true" in the chosen DL domain, then it is true in the boolean logic, and similarly for false.
- *Compositionality*: the translation function should preserve the structural properties, e.g., (the translation of) negation should compose with conjunction and disjunction, and (the translation of) conjunction and disjunction should satisfy the usual properties of idempotence, commutativity, and associativity.
- *Shadow-lifting*: the resulting functions should have partial derivatives that can characterise the idea of gradual improvement in training [27]. For example, a translation of a conjunction should evaluate to a higher value if the value of one of its conjuncts increases.

Unfortunately, none of the existing DLs satisfies all of these requirements [24, 27]. Therefore, future tools and compilers such as Vehicle may need to provide support for incorporating a range of DLs for different scenarios.

This conclusion brings to the forefront the need for a generic framework in which logical and geometric properties of different DLs can be formalised and proven. In this paper, we propose a unified formalisation of DLs to lay down the ground for the development of a reliable neural network verification tool. For that purpose, we will build on top of previous work that has already proposed a common presentation of DLs [24]. In order to handle the verification of translation from properties to loss functions, we use the Coq proof assistant in which numerous formalisations of logics and programming languages have been carried out. In addition, the formalisation of the properties of DLs also requires a good library

support for algebra (to handle the structural properties of DLs) as well as analysis (to handle shadow-lifting), a task for which the Mathematical Components libraries (hereafter, MathComp [25]) seem well fitted.

Our contributions in this paper are as follows:

- We explain how to encode known DLs in a single generic syntax using Coq, taking advantage of dependent types and building on known techniques for logic embedding (such as intrinsic typing). The formalisation is comprehensive and extensible for future use.
- We demonstrate how to use the MathComp libraries for our purpose, which includes reusable lemmas that we had to newly develop.
- As a result we are able to find and fix errors in the literature. The most prominent missing results were: soundness of STL and missing parts of the shadow-lifting proofs, both of which appear as original results in this paper.

The paper proceeds as follows. Section 2 provides further background information about property-guided training and DLs. Section 3 explains how one can define DLs in Coq using a generic encoding, including a translation function producing the semantics. Section 4 focuses on the formalisation of logical properties and soundness of DLs. In Section 5, we demonstrate the formal verification of the shadow-lifting properties of DLs. We discuss related work and conclude in Section 6.

2 Background

2.1 Property-guided training, by means of an example

Neural network properties

Given a neural network $N : \mathbb{R}^m \to \mathbb{R}^n$, the verification property usually takes the form of a Hoare triple $\forall \mathbf{x} \in \mathbb{R}^m.\mathcal{P}(\mathbf{x}) \longrightarrow \mathcal{S}(\mathbf{x})$, where $\mathcal{P}$ and $\mathcal{S}$ can be arbitrary properties obtained by using variables $\mathbf{x} \in \mathbb{R}^m$, constants, vector, arithmetic operations, $\leq$, $=$, $\wedge$, $\vee$, and $\neg$. Additionally, $\mathcal{S}$ may contain the neural network N as a function.

▶ **Example 1** (Properties of neural networks)**.** Given a neural network N and a vector $\mathbf{v}$, consider the specification that requires that for all inputs $\mathbf{x}$ that are within ϵ distance from $\mathbf{v}$, the output of $N(\mathbf{x})$ should not deviate by more than δ from $N(\mathbf{v})$:

$$\forall \mathbf{x}.|\mathbf{x} - \mathbf{v}|_{L_\infty} \leq \epsilon \Rightarrow |N(\mathbf{x}) - N(\mathbf{v})|_{L_\infty} \leq \delta.$$

This property is known as ϵ-δ-*robustness* [9]. It can be used to avoid misclassifying images when only a few pixels are perturbed. This particular example uses the L_∞ *norm*: $|\mathbf{x}-\mathbf{y}|_{L_\infty} \stackrel{\text{def}}{=} \max_{i\in\{0,\ldots,n-1\}}([\mathbf{x}]_i - [\mathbf{y}]_i)$, where $[\mathbf{x}]_i$ stands for the ith element of $\mathbf{x}$.

Unfortunately, as demonstrated by Fischer et al. [16], even most accurate neural networks fail even the most natural verification properties, such as ϵ-δ-robustness. This motivated the search for better ways to train the networks.

Property-guided training

Methods for property-guided training have received considerable attention in the AI literature, as the survey [18] shows. We will only illustrate the method that was suggested by Fischer et al. [16], and refer the reader to the survey for more examples.

▶ **Example 2** (Generating a loss function from a logical property [16])**.** Recall that standard supervised learning trains a neural network N with trainable parameters θ to optimise the objective $\min_\theta \mathcal{L}(\mathbf{x}, \mathbf{y})$, for the loss function $\mathcal{L} : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$. Generally, $\mathcal{L}$ measures the difference between the network's output and the given data for each input point. Examples of $\mathcal{L}$ are cross-entropy loss or mean squared error. But now we want to train the neural network to satisfy any arbitrary property $\forall \mathbf{x}.\mathcal{P}(\mathbf{x}) \longrightarrow \mathcal{S}(\mathbf{x})$. For this, we replace the above optimisation objective with

$$\min_\theta \left[\max_{\mathbf{x} \in \mathbb{H}_{\mathcal{P}(\mathbf{x})}} \mathcal{L}_{\mathcal{S}}(\mathbf{x}) \right]$$

where $\mathbb{H}_{\mathcal{P}(\mathbf{x})} \subseteq \mathbb{R}^m$ refines the type $\mathbb{R}^m$ to a subset for which the property $\mathcal{P}$ holds, and $\mathcal{L}_{\mathcal{S}} : \mathbb{R}^m \to \mathbb{R}$ is obtained by applying a suitable *interpretation function* for $\mathcal{S}$.

We omit the exact details of how such optimisation algorithms are defined: they are known and can be found in a suitable machine learning tutorial, for example [19]. Intuitively, the optimisation algorithm will search for $\mathbf{x} \in \mathbb{H}_{\mathcal{P}(\mathbf{x})}$ such that $\mathbf{x}$ maximises the loss $\mathcal{L}_{\mathcal{S}}(\mathbf{x})$, in order to train the neural network parameters θ to minimise that loss. Concretely, if the property is ϵ-δ-robustness, it will look for the worst perturbation of $\mathbf{v}$ that violates the property, and will optimise the neural network to classify that bad example correctly.

Differentiable logics for loss functions

In the above example, we did not explain how to define the interpretation function $\mathcal{L}_{\mathcal{S}}$ for an arbitrary property $\mathcal{S}$; we need differentiable logics for that purpose. Fischer et al. [16] proposed one such interpretation function – called *the differentiable logic DL2*, standing for "Deep Learning with Differentiable Logics".

▶ **Example 3** (Loss functions from properties in a fuzzy logic)**.** Taking the properties from Example 1, by the Fischer et al. method we must be able to interpret the right-hand side of the implication, i.e., $|N(\mathbf{x}) - N(\mathbf{v})|_{L_\infty} \leq \delta$, given concrete values for ϵ, δ, a concrete vector $\mathbf{v}$, neural network N, and a suitable definition of the L_∞ norm. For example, interpretation for our property in STL [27] is given by $[\![|N(\mathbf{x}) - N(\mathbf{v})|_{L_\infty} \leq \delta]\!]_{\mathrm{STL}} = \delta - |[\![N(\mathbf{x})]\!] - [\![N(\mathbf{v})]\!]|_{L_\infty}$. On the left-hand side, the L_∞ distance between vectors as well as N are defined in the syntax of STL; on the right-hand side, they are given by real vector arithmetic operations. Example 8 will make the relation between syntax and interpretation clear. The obtained function can be used directly for training neural networks.

We next consider our choices of DLs in details.

2.2 Differentiable logics

Ślusarz et al. [24] suggest a common syntax for all DLs, calling it the *logic of differentiable logics (LDL)*, and subsequently obtain different DLs via different interpretation functions. In the following, we summarize the syntactic and semantic features of DLs following this formulation; minor modifications will be discussed as we introduce them.

LDL syntax

LDL's syntax consists of types and expressions (Fig. 1). Types are given by booleans, reals, vectors, indices, and a function type Fun n m; expressions are given by real numbers, vectors, vector indices, lookup operations, and functions that take real vectors as inputs. Formulae

type $\ni t ::= \text{Bool} \mid \text{Index } n \text{ for } n \in \mathbb{N} \mid \text{Real} \mid \text{Vec } n \mid \text{Fun } n\ m \text{ for } n, m \in \mathbb{N}$

$$
\begin{array}{rlll}
\text{exprInd} & \ni i & ::= & i \in \mathbb{N} \\
\text{exprR} & \ni r, r_1, r_2 & ::= & r \in \mathbb{R} \mid [v]_i \\
\text{exprFun} & \ni f & ::= & f \in \mathbb{R}^n \to \mathbb{R}^m \\
\text{exprVec} & \ni v & ::= & v \in R^n \mid f\ v
\end{array}
\qquad
\begin{array}{rlll}
\text{exprB} & \ni p, p_0, \ldots, p_n & ::= & \mathit{True} \mid \mathit{False} \\
 & & \mid & r_1 \le r_2 \\
 & & \mid & r_1 \neq r_2 \\
 & & \mid & \bigwedge_M (p_0, \ldots, p_M) \\
 & & \mid & \bigvee_M (p_0, \ldots, p_M) \\
 & & \mid & \neg p
\end{array}
$$

Figure 1 Types and expressions of LDL.

are formed either via applying predicates $\le, =$ to real expressions, by boolean values, or using logical connectives $\vee, \wedge, \neg$. Because STL by Varnai et al. [27] lacks associativity, conjunction and disjunction are defined as n-ary connectives so that they are the same for all DLs. Further, implication is not present in the syntax: that is due to the n-ary nature of the other connectives, which do not always allow for the implication of classical logic. Any DL with associative conjunction and disjunction will admit implication $b_1 \Rightarrow b_2$ to be defined as $\neg b_1 \vee b_2$.

We forgo the originally included quantifiers, lambda, and let expressions to obtain a simpler core language in which the three properties of interest – soundness, compositionality, and differentiability – can be studied.

Obtaining DLs via interpretation functions

To define a DL, one defines an interpretation function $[\![\cdot]\!]_{\text{DL}}$ that, given an expression in LDL, returns a function on real numbers. We introduce all DLs in a generic way and use the meta-notation $[\![\cdot]\!]_{\text{DL}}$, to refer to a range of interpretation functions, with $\text{DL} \in \{\text{Gödel}, \text{Łukasiewicz}, \text{Yager}, \text{product}, \text{DL2}, \text{STL}\}$. The boolean interpretation function $[\![\cdot]\!]_{\text{B}}$ is the obvious logical interpretation of boolean formulas, which will be useful for proving soundness later.

Table 1 shows the interpretation of all DLs. First are the four DLs based on well-known fuzzy logics: Gödel, Łukasiewicz [28], Yager, and product [26]. All fuzzy logics have the interpretation domain of $[0, 1] \subset \mathbb{R}$. Other logics have different domains: DL2 [16] has the interpretation domain $(-\infty, 0]$, and STL [27] the domain $(-\infty, +\infty)$.

The binary predicates $\le$ and $=$ are defined in a way that ensures that they are interpreted within the chosen real interval for the given DL. The definitions of logical connectives $\bigvee_M$, $\bigwedge_M$, and $\neg$ are taken directly from the related papers that define the given DLs. Note that we reformulate $\bigvee_M$ and $\bigwedge_M$ for all DLs as n-ary connectives, however, only STL had n-ary connectives originally.

2.3 Properties of DLs

Soundness

There is no consensus in the DL literature on how or whether to state soundness: for example, STL came without any soundness statement. For the sake of generic formalisation of all DLs, we propose the following definition of soundness, which generalises soundness as defined in DL2 and fuzzy logics [16, 26].

Table 1 Interpretation function $[\![\cdot]\!]_{\mathrm{DL}}$, which extends naturally to sequences of expressions as well as interpretation function $[\![\cdot]\!]_{\mathrm{B}}$, which is a structural interpretation of boolean formulas.
†: Negation is undefined in the sense that it is not defined as a structural transformation of the syntax of formulas. It is implemented at the level of atomic comparison only, and negation on composite formulas is provided as syntactic sugar [16, Sect. 3]. For example, consider $[\![3 = 3]\!]_{\mathrm{DL2}} = 0$. In DL2, $[\![\neg(3 = 3)]\!]_{\mathrm{DL2}}$ is not defined in term of $[\![3 = 3]\!]_{\mathrm{DL2}} = 0$, but DL2 provides an interpretation for the symbol $\neq$ separately.

	$[\![\bigwedge_M s]\!]$	$[\![\bigvee_M s]\!]$	$[\![\neg e]\!]$
Gödel	$\min [\![s]\!]_{\mathrm{G}}$	$\max [\![s]\!]_{\mathrm{G}}$	$1 - [\![e]\!]_{\mathrm{G}}$
Łukasiewicz	$\max\left[\sum_{a\in[\![s]\!]_{\mathrm{Ł}}} a - \lvert s\rvert + 1, 0\right]$	$\min\left[\sum_{a\in[\![s]\!]_{\mathrm{Ł}}} a, 1\right]$	$1 - [\![e]\!]_{\mathrm{Ł}}$
Yager	$\max\left[1 - \left(\sum_{a\in[\![s]\!]_{\mathrm{Y}}} (1-a)^p\right)^{1/p}, 0\right]$	$\min\left[\left(\sum_{a\in[\![s]\!]_{\mathrm{Y}}} a^p\right)^{1/p}, 1\right]$	$1 - [\![e]\!]_{\mathrm{Y}}$
product	$\prod_{a\in[\![s]\!]_{\mathrm{P}}} a$	fold $(\lambda x\ y\ .\ x + y - xy)\ 0\ [\![s]\!]_{\mathrm{P}}$	$1 - [\![e]\!]_{\mathrm{P}}$
DL2	$\sum_{a\in[\![s]\!]_{\mathrm{DL2}}} a$	$(-1)^{\lvert s\rvert+1} \cdot \prod_{a\in[\![s]\!]_{\mathrm{DL2}}} a$	undefined†
STL	$and_S\ [\![s]\!]_{\mathrm{STL}}$	$or_S\ [\![s]\!]_{\mathrm{STL}}$	$-[\![e]\!]_{\mathrm{STL}}$
Bool	$\bigwedge_M [\![s]\!]_{\mathrm{B}}$	$\bigvee_M [\![s]\!]_{\mathrm{B}}$	$\neg[\![e]\!]_{\mathrm{B}}$

	$[\![e_1 = e_2]\!]$	$[\![e_1 \le e_2]\!]$	$[\![True]\!]$	$[\![False]\!]$
fuzzy	if $[\![e_1]\!] = -[\![e_2]\!]$ then $[\![e_1]\!] = [\![e_2]\!]$ else $\max\left[1 - \left\lvert\frac{[\![e_1]\!]-[\![e_2]\!]}{[\![e_1]\!]+[\![e_2]\!]}\right\rvert, 0\right]$	if $[\![e_1]\!] = -[\![e_2]\!]$ then $[\![e_1]\!] \le [\![e_2]\!]$ else $\max\left[1 - \max\left[\frac{[\![e_1]\!]-[\![e_2]\!]}{[\![e_1]\!]+[\![e_2]\!]}, 0\right], 0\right]$	1	0
DL2	$-\lvert[\![e_2]\!]_{\mathrm{DL2}} - [\![e_1]\!]_{\mathrm{DL2}}\rvert$	$-\max\left[[\![e_1]\!]_{\mathrm{DL2}} - [\![e_2]\!]_{\mathrm{DL2}}, 0\right]$	0	$-\infty$
STL	$-\lvert[\![e_2]\!]_{\mathrm{STL}} - [\![e_1]\!]_{\mathrm{STL}}\rvert$	$[\![e_2]\!]_{\mathrm{STL}} - [\![e_1]\!]_{\mathrm{STL}}$	$+\infty$	$-\infty$
Bool	$[\![e_1]\!]_{\mathrm{B}} = [\![e_2]\!]_{\mathrm{B}}$	$[\![e_1]\!]_{\mathrm{B}} \le [\![e_2]\!]_{\mathrm{B}}$	*True*	*False*

$$and_S\ [a_1, \ldots, a_M] = \begin{cases} \dfrac{\sum_i a_{\min} e^{\tilde{a_i}} e^{\nu \tilde{a_i}}}{\sum_i e^{\nu \tilde{a_i}}} & \text{if } a_{\min} < 0 \\ \dfrac{\sum_i a_i e^{-\nu \tilde{a_i}}}{\sum_i e^{-\nu \tilde{a_i}}} & \text{if } a_{\min} > 0 \\ 0 & \text{if } a_{\min} = 0 \end{cases} \quad \text{where} \quad \begin{array}{rcl} \nu & \in & \mathbb{R}^+ \text{ (constant)} \\ a_{\min} & = & \min [a_1, \ldots, a_M] \\ \tilde{a_i} & = & \dfrac{a_i - a_{\min}}{a_{\min}} \end{array}$$

or_S is analogous to and_S

▶ **Definition 4** (Soundness). *Given a DL, an expression e, and a boolean value $b \in \{True, False\}$, the DL is sound if*

$$[\![e]\!]_{DL} = [\![b]\!]_{DL} \implies [\![e]\!]_B = b.$$

Note that not all DLs are sound. For example, one of the oldest fuzzy logics by Łukasiewicz [28] is known to be unsound. Table 2 summarises all known soundness results. Note that prior to this paper, soundness of STL was not known. Here, we obtain the result with some restrictions, see Sect. 4.

Compositionality

We define idempotence, associativity, and commutativity of interpretation functions for $\bigwedge_M$ and analogously $\bigvee_M$:

▶ **Definition 5** (Commutativity, idempotence, and associativity of $\bigwedge_M$). *Given a DL, the interpretation function of conjunction is commutative if for any permutation π of the integers $i \in \{1, \ldots, M\}$ we have*

$$\left[\!\!\left[\bigwedge_M (p_0, \ldots, p_M)\right]\!\!\right]_{DL} = \left[\!\!\left[\bigwedge_M (p_{\pi(0)}, \ldots, p_{\pi(M)})\right]\!\!\right]_{DL}.$$

Table 2 Properties of the different DLs formalised in this paper [29]. We distinguish previously known proofs that we mechanise from previously known results published with incomplete or semi-formal proofs (yellow) and new results (orange).
†: For DL2 and STL, we prove soundness of the negation-free fragment of LDL; negation is undefined for DL2, and STL is not sound for the full fragment.

Properties:	Negation	Idempotence	Commutat.	Associativ.	Soundness	Shadow-lifting
Gödel	yes	yes	yes	yes	yes	no
Łukasiewicz	yes	no	yes	yes	no	no
Yager	yes	no	yes	yes	no	no
product	yes	no	yes	yes	yes	yes
DL2	no	no	yes	yes	yes†	yes
STL	yes	yes	yes	no	yes†	yes

It is idempotent and associative if we have

$$\left[\!\left[\bigwedge_M (p,\ldots,p) \right]\!\right]_{DL} = [\![p]\!]_{DL},$$
$$\left[\!\left[\bigwedge_M \left(\bigwedge_M (p_0,p_1), p_2 \right) \right]\!\right]_{DL} = \left[\!\left[\bigwedge_M \left(p_0, \bigwedge_M (p_1,p_2) \right) \right]\!\right]_{DL}.$$

Table 2 shows which DLs satisfy which logical properties. Finally, as already illustrated in Sect. 2.1, negation can be problematic in some DLs. For example, DL2 does not give a direct interpretation for negation, as its domain is asymmetric. We will see in Sect. 4 that negation also causes problems with the soundness of STL.

Differentiability

Varnai et al. [27] introduce three properties in this category: weak smoothness, scale-invariance, and shadow-lifting. The latter is the most important as it accounts for gradual improvement in training. We only consider shadow-lifting here as it is the most complex of those properties and leave the remaining properties to future work.

▶ **Definition 6** (Shadow-lifting property [27]). *The DL satisfies the shadow-lifting property if, for any* $[\![p]\!]_{DL} \neq 0$*:*

$$\left. \frac{\partial \left[\!\left[\bigwedge_M (p_0,\ldots,p_i,\ldots,p_M) \right]\!\right]_{DL}}{\partial [\![p_i]\!]_{DL}} \right|_{p_j = p \text{ where } i \neq j} > 0$$

holds for all $0 \leq i \leq M$*, where* ∂ *denotes partial differentiation.*

Notice that classical conjunction does not satisfy the property of shadow-lifting: no matter how "true" the value of p_2 is, if p_1 is false, then $p_1 \wedge p_2$ will remain false. Likewise, all DLs that use min or max to define conjunction will fail shadow-lifting.

Shadow-lifting was originally defined for conjunction only, as STL had no disjunction. In our formalisation, we could, in principle, extend shadow-lifting to disjunction. However, we left this incremental extension for future work.

Summary of results

Table 2 summarises all properties covered in our Coq formalisation and highlights the ones for which we provide original proofs. In our development, we provided several missing results, most prominently, the soundness of STL and missing parts of the shadow-lifting proofs. Note that the formalisation further revealed some errors:

▶ **Example 7** (Discrepancies in pen and paper proofs)**.** While doing the Coq formalisation, we found two sources of errors in our pen and paper proofs [24] as well as in [27]. Firstly, the work in [24] concerned completing results of Table 2 in the uniform notation of LDL. Many proofs were analogous and it was easy to overlook the rare cases when the proof could not be completed by analogy with existing proofs. For example, we tried to prove the soundness of Yager by analogy with other fuzzy logics overlooking that Yager is not sound. Indeed Yager is a generalisation of Łukasiewicz logic which in itself is not sound. The Coq formalisation revealed all such errors.

The second source of errors came from extension of fuzzy logics with comparison operators. The interpretation of comparisons needed to be scaled between 0 and 1, and it seemed obvious that the scaling was done correctly. Therefore we did not provide any proofs concerning the interval properties of these operations. In contrast the soundness proofs in Coq required us to prove that the fuzzy interpretation functions always return values within the interval $[0, 1]$.

Thirdly, while a sketch of the shadow-lifting proof was provided in [27], it was incomplete and did not mention either the existence of other cases as well as the reliance of the proof on L'Hôpital's rule.

No DL satisfies all desirable properties – for example, Gödel is sound, idempotent, associative, and commutative, but it is not shadow-lifting. On the other hand, Łukasiewicz is not sound, STL not associative, and while DL2 is sound and shadow-lifting, it fails idempotence, and its negation is not compositional because it is not structural. Varnai et al. [27] have proven that it is impossible for any DL to be idempotent, associative, and shadow-lifting at the same time.

When one has to make a choice of a DL, different considerations may influence that choice. Soundness and shadow-lifting are strictly desirable, thus Gödel, Łukasiewicz and Yager are probably less desirable than the rest, even if some of them have nice logical properties. However, given soundness and shadow-lifting, the choice between logical properties is less clear. For example, one can imagine a scenario when the specification language avoids negation, and in a style of substructural logics, treats differently p and $p \wedge p$ and thus sacrifices idempotence; in this case, DL2 may provide an ideal translation function.

3 An encoding of DLs in Coq

As discussed, LDL aims at defining all DLs in a generic and extendable way, using uniform syntactic conventions. In this section, we start by highlighting the generic features of our formalisation.

3.1 Encoding of the syntax of types and expressions

The encoding of the LDL types is the matter of declaring the following inductive type:

```
Inductive flag := def | undef.

Inductive ldl_type :=
  Bool_T of flag | Index_T of nat | Real_T | Vector_T of nat | Fun_T of nat & nat.
```

This corresponds to the informal syntax of Fig. 1 with the difference that we refine the Boolean type with a (`flag`) to signify whether negation is defined in the logic.

As for LDL expressions, their encoding is displayed in Fig. 2. It is an inductive type indexed by `ldl_type` so that the resulting syntax is intrinsically-typed: one cannot write ill-typed expressions. The Coq inductive type matches the informal syntax already explained

in Fig. 1. Real expressions (line 6) use a type `R` of type `realType` coming from MathComp-Analysis [1] that represents real numbers. Boolean expressions (line 7) use the native Coq type `bool`. Indices embed an *ordinal* from MathComp (line 8). More specifically, `'I_n` is the type of natural number smaller than `n`. Similarly, vectors just reflect MathComp tuples (line 9). For defining n-ary connectives `ldl_and` and `ldl_or`, we use polymorphic lists (of type `seq`). Yet, to ease reading, we will use notation such as `a `/\ b` to denote binary conjunction in the following. For a generic definition of the syntax, we need to allow for the case of DLs in which negation is not defined (in fact, DL2). The additional argument `Bool_T_def` in the constructor `ldl_not` (line 12) signifies that the negation is defined. The constructor `ldl_cmp` (line 13) is for binary comparison operators over the real numbers; hereafter, we will use notations such as `` `<= `` for the comparison corresponding to $\leq$ instead of "`ldl_cmp cmp_le`" to ease reading. As for the last constructors, they are respectively for functions, their application, and lookups, as per Fig. 1.

```
1  Definition Bool_T_undef := Bool_T undef.
2  Definition Bool_T_def := Bool_T def.
3  Inductive comparison := cmp_le | cmp_eq.
4
5  Inductive expr : ldl_type -> Type :=
6   | ldl_real   : R -> expr Real_T
7   | ldl_bool   : forall p, bool -> expr (Bool_T p)
8   | ldl_idx    : forall n, 'I_n -> expr (Index_T n)
9   | ldl_vec    : forall n, n.-tuple R -> expr (Vector_T n)
10  | ldl_and    : forall x, seq (expr (Bool_T x)) -> expr (Bool_T x)
11  | ldl_or     : forall x, seq (expr (Bool_T x)) -> expr (Bool_T x)
12  | ldl_not    : expr Bool_T_def -> expr Bool_T_def
13  | ldl_cmp    : forall x, comparison -> expr Real_T -> expr Real_T -> expr (Bool_T x)
14  | ldl_fun    : forall n m, (n.-tuple R -> m.-tuple R) -> expr (Fun_T n m)
15  | ldl_app    : forall n m, expr (Fun_T n m) -> expr (Vector_T n) -> expr (Vector_T m)
16  | ldl_lookup : forall n, expr (Vector_T n) -> expr (Index_T n) -> expr Real_T.
```

Figure 2 LDL syntax in Coq.

3.2 Encoding of the interpretation function

We now proceed to the translation function that interprets the syntax. Types are mapped to their obvious semantics:

```
Definition type_translation (t : ldl_type) : Type :=
match t with
| Bool_T x => R
| Real_T => R
| Vector_T n => n.-tuple R
| Index_T n => 'I_n
| Fun_T n m => n.-tuple R -> m.-tuple R
end.
```

In particular, booleans are mapped to `R` of type `realType`, the type of real numbers. This translation accommodates the many interpretations of the DLs: the domain $[0,1]$ for fuzzy logic, $(-\infty,0]$ for DL2, and $(-\infty,+\infty)$ for STL. For DL2 and STL, we also provide an alternative translation function `ereal_type_translation` that maps booleans to real numbers extended with $-\infty$ and $+\infty$, i.e., the type `\bar R` of extended real numbers as provided by MathComp-Analysis. We use an invariant to restrict the range of the interpretation function accordingly.

Each logic requires a separate interpretation function (as explained in Table 1). Here, we only show an excerpt of the translation function for STL, namely the cases for conjunction (constructor `ldl_and`), negation (notation `` `~ ``), and comparison (notation `` `<= ``) of STL in Fig. 3.

```
Fixpoint stl_translation {t} (e : expr t) : type_translation t :=
  match e in expr t return type_translation t with
  | ldl_and _ (e0 :: s) => let A     := map stl_translation s in
                           let a0    := stl_translation e0 in
                           let a_min := \big[minr/a0]_(i <- A) i in
                           if a_min < 0 then stl_and_lt0 (a0 :: A) else
                           if a_min > 0 then stl_and_gt0 (a0 :: A) else
                           0
  | `~ E1               => - {[ E1 ]}
  | E1 `<= E2           => {[ E2 ]} - {[ E1 ]}
  ... (* see [29] for omitted connectives *)
  end where "{[ e ]}" := (stl_translation e).
```

Figure 3 Excerpt of the semantics of STL: conjunction, negation, and comparison. (See Fig. 4 for intermediate definitions `stl_and_lt0` and `stl_and_gt0`.)

The case for conjunction of STL is the most complex in our formalisation because dealing formally with it requires the theories of exponentiation (`expR`), big sums (`\sum`), inverses (`^-1`), and minima (`minr`). To reduce the clutter, we define the cases for $a_{\min} > 0$ and $a_{\min} < 0$ separately as `stl_and_gt0` and `stl_and_lt0` reproduced in Fig. 4. This will allow us to state intermediate lemmas about sub-expressions.

```
Definition sumR a := \sum_(i <- a) i.
Definition min_dev {R : realType} (x:R) (a:seq R) : R :=
  let r := \big[minr/x]_(i <- a) i in (x - r) * r^-1.

Definition stl_and_gt0 (a : seq R) :=
  sumR (map (fun x => x * expR(-nu * min_dev x a)) a) *
  (sumR (map (fun x => expR(-nu * min_dev x a)) a))^-1.

Definition stl_and_lt0 (a : seq R) :=
  sumR (map (fun x => (\big[minr/x]_(i <- a) i) *
    expR (min_dev x a) * expR(nu * min_dev x a)) a) *
  (sumR (map (fun x => expR(nu * min_dev x a)) a))^-1.
```

$$a_{\min} = \min\,[a_1, \ldots, a_M]$$
$$\tilde{a_i} = \frac{a_i - a_{\min}}{a_{\min}}$$

$\nu \in \mathbb{R}^+$	(constant)
$\frac{\sum_i a_i e^{-\nu \tilde{a_i}}}{\sum_i e^{-\nu \tilde{a_i}}}$	(case $a_{\min} > 0$)
$\frac{\sum_i a_{\min} e^{\tilde{a_i}} e^{\nu \tilde{a_i}}}{\sum_i e^{\nu \tilde{a_i}}}$	(case $a_{\min} < 0$)

Figure 4 Intermediate definitions to define the conjunction of STL. (The right subfigure reproduces part of Table 1 for reading convenience.)

As a first application of our encoding, we can already formalise our running example, with the advantage of encoding it once for all DLs:

▶ **Example 8.** Taking the interpretation task of Example 3, we first give a suitable definition for L_∞ norm (`ldl_norm_infty`) and vector subtraction (`ldl_vec_sub`). One only needs to call the defined interpretation function to encode the loss function of Example 3:

```
Context (eps delta : @expr R Real_T) (f : @expr R (Fun_T n.+1 m.+1))
        (v : @expr R (Vector_T n.+1)) (x : @expr R (Vector_T n.+1)).

Definition eps_delta_robust : expr Bool_T_undef :=
  ldl_lookup
    (ldl_app (ldl_norm_infty m) (ldl_vec_sub (ldl_app f x) (ldl_app f v)))
    (ldl_idx ord0) `<= delta.
```

4 Logical properties and soundness of DLs

4.1 Logical properties of DLs

The logical properties of DLs are idempotence, commutativity, and associativity. Not all DLs have the same properties as we saw earlier (Table 2). Proving the logical properties essentially amounts to showing that the semantic interpretation does have them. For example, the conjunction of DL2 being interpreted as addition on reals inherits its associativity directly from the properties of real numbers, and as a consequence, its proofs are one-liners, e.g.:

```
Lemma dl2_andA (e1 e2 e3 : expr Bool_T_undef) :
  [[ e1 `/\ (e2 `/\ e3) ]]_dl2 = [[ (e1 `/\ e2) `/\ e3 ]]_dl2.
Proof. by rewrite /=/sumR ?big_cons ?big_nil !addr0 addrA. Qed.
```

In contrast, for Yager and STL, the proofs are more demanding. For example, the associativity for Yager, though stated analogously,

```
Theorem Yager_andA (e1 e2 e3 : expr Bool_T_def) :
   0 < p ->  [[ (e1 `/\ e2) `/\ e3]]_Yager = [[ e1 `/\ (e2 `/\ e3) ]]_Yager.
```

consists of about 100 lines of code. This is because in this case, the interpretation relies on the power function of MathComp-Analysis whose properties are more technical. Yet, we could put the automatic tactics available with MathComp such as `lra` [22, 23] to good use.

4.2 Soundness of DLs

We now address the topic of formalising the soundness results of Table 2.

Soundness for closed interval DLs

For fuzzy DLs and, more generally, closed interval DLs there is a clear consensus on how to define soundness: we generalised it in Definition 4. It boils down to taking the least and greatest elements in the given real interval as interpretations for *False* and *True*, respectively. In Coq, the statement of soundness for Gödel and product is as follows:

```
Lemma soundness (e : expr Bool_T_def) b :
  [[ e ]]_ 1 = [[ ldl_bool _ b ]]_ 1 -> [[ e ]]_B = b.
```

This is a direct paraphrase of the pencil-and-paper Definition 4. The proofs proceed by induction and require inversion lemmas, which we will discuss later in this section.

Soundness for DLs with open intervals

When a DL's domain of interpretation is given by an open interval, which is the case for DL2 and STL, there is no clear consensus in the literature on defining or proving soundness. We will illustrate the problems that arise using STL and following previous work [24]. The first question is how to state soundness. The easiest choice is to simply add $-\infty$ and $+\infty$ as constants to the domain, and keep the soundness statement of Definition 4. However, because no formula in the language evaluates to $-\infty$ or $+\infty$, such a soundness proof is vacuous. Note that Definition 4 did not cause this problem for fuzzy DLs because there were formulae in the language that evaluated to bottom and top values: take for example $[\![3 = 3]\!]_{\mathrm{P}} = 1$.

Alternatively, one may keep the open interval intact and simply re-define soundness in terms of intervals: if the interpretation of the formula e is greater or equal to 0, then $[\![e]\!]_{\mathrm{B}} = \mathit{True}$ else $[\![e]\!]_{\mathrm{B}} = \mathit{False}$. However, this solution triggers a different problem: negation is no longer sound. Indeed, if $[\![3 = 3]\!]_{\mathrm{STL}} = 0$ means the formula is true, then the same can be said about $[\![\neg(3 = 3)]\!]_{\mathrm{STL}} = 0$.

One could think of a solution excluding 0 from the interval $(-\infty, +\infty)$ altogether, but that complicates the interpretation of comparisons and creates a point in the interval at which the resulting function is not differentiable, which damages shadow-lifting.

Coq formalisation for logics with open intervals

For the reasons explained above, we remove negation from STL and use intervals to define the truth:

```
Definition is_stl b (x : R) := if b then x >= 0 else x < 0.
```

This results in the following soundness statement:

```
Lemma stl_soundness (e : expr Bool_T_undef) b :
  is_stl b (nu.-[[ e ]]_stl) -> [[ e ]]_B = b.
```

The flag `Bool_T_undef` in `(e : expr Bool_T_undef)` signifies that the proof omits the case that uses negation. We write `nu.-[[ e ]]_stl` as notation for interpretation of STL (and similar notation for the remaining DLs).

The soundness proof proceeds by induction on the structure of the interpretation function. Because of the extensive use of dependent types in our formalisation we need a custom dependent induction principle. The most interesting cases are those for conjunction and disjunction, which need special inversion lemmas. Here is one example:

```
Lemma stl_nary_inversion_andE1 (s : seq (expr Bool_T_undef)) :
  is_stl true (nu.-[[ ldl_and s ]]_stl) ->
    forall i, i < size s -> is_stl true (nu.-[[ nth (ldl_bool pos false) s i ]]_stl).
```

Our formalisation faced a minor technical problem: if we are to comply with the generic DL syntax defined in Fig. 1, we need to interpret constants *True* and *False* present in the language. We therefore propose two alternative interpretations for DL2 and STL: one that works on extended reals (with added constants $-\infty$, $+\infty$) and maps *True* and *False* to the top and bottom elements of the respective domains, and one that resolves this discrepancy by choosing arbitrary interpretations for *True* and *False* that satisfy all the properties of interest for our study. In the latter case, for DL2 we choose to interpret *True* as 0 and *False* as -1, and for STL we choose to interpret *True* as 1 and *False* as -1. In all these four cases we show that the resulting logic satisfies the soundness property stated above. Adding $-\infty$ and $+\infty$ has repercussions when proving the geometric properties of the logics, as we show later in Sect. 5. If it were not for considerations of using the generic syntax for all DLs, *True* and *False* could be removed from the STL syntax altogether, without damaging the main results.

Lessons learnt

Soundness for DLs with open intervals was the first real challenge that this formalisation faced. Having no plausible solution in the field, being able to use Coq to experiment with different soundness statements and see their effect on proofs was extremely rewarding. Overall, we proved three different versions of STL soundness (one for "vacuous proofs", which we do not present here); and we intend to use this formalisation to experiment further with STL. In particular, finding an alternative approach to negation, e.g., using "approximate 0", is now within our reach. The currently presented approach is the first proof of soundness for any fragment of STL, it already covers formalisation of problems such as the ϵ-δ-robustness; and we attribute this intermediate success to the assistance of the Coq formalisation.

5 Differentiability: shadow-lifting

It was Varnai et al. who provided for STL the pencil-and-paper proof of shadow-lifting [27, Sect. V] (along with the definition of the STL conjunction). This section formalises this result and actually completes it since the original proof only covers one of the two non-trivial cases. The main technical aspect of the proof is high-school level mathematics: an application of L'Hôpital's rule, which was not yet available in MathComp-Analysis.

DL2 and the product DL also trivially enjoy shadow-lifting. In the following, we will therefore start by formalising the latter DLs, then formalising L'Hôpital's rule, and finally provide an overview of the missing part of Varnai et al.'s proof of shadow-lifting for STL. Note that the logics Gödel, Łukasiewicz, Yager fail shadow-lifting as they are not differentiable everywhere, due to their use of min or max to define conjunction.

5.1 formalisation of shadow-lifting

As seen in Sect. 2, shadow-lifting is defined in terms of partial derivatives, for which there was however no theory yet in MathComp-Analysis. They can be easily defined on the model of derivatives [1, file `derive.v`]. First, we define *error vectors* as row vectors (type `'rV[R]_k` where `R` is some ring) that are 0 everywhere except at one coordinate `i`:

```
Definition err_vec {R : ringType} (i : 'I_n.+1) : 'rV[R]_n.+1 :=
  \row_(j < n.+1) (i == j)%:R.
```

The notation `%:R` injects a natural number into a ring; note that here the boolean equality (notation: `==`) is implicitly coerced to a natural number. Then, given a function `f` that takes as input a row vector, we define a function `partial` that given a row vector `a` and an index `i` returns the limit $\lim_{\substack{h\to 0\\ h\neq 0}} \frac{\texttt{f(a+}h\texttt{*err\_vec i)-f(a)}}{h}$. Put formally:

```
Definition partial {R} {n} (f : 'rV[R]_n.+1 -> R) (a : 'rV[R]_n.+1) i :=
  lim (h^-1 * (f (a + h *: err_vec i) - f a) @[h --> 0^']).
```

In this syntax, `0^'` represents the deleted neighborhood of `0`, and `lim g @ F` represents the limit of the function `g` at the filter `F` [2]. The notation `*:` represents scaling but is equivalent to the multiplication of real numbers here. Hereafter, we use the Coq notation `d f '/d i` for `partial f i`.

Using partial derivatives, the definition of shadow-lifting (Definition 6) translates directly into Coq:

```
Definition shadow_lifting {R : realType} n (f : 'rV_n.+1 -> R) :=
  forall p, p > 0 -> forall i, ('d f '/d i) (const_mx p) > 0.
```

The `const_mx` function comes from MathComp's matrix theory and represents a matrix where all coefficients are the given constant; we use it to implement the restriction "$p_j = p$" seen in Definition 6.

5.2 Shadow-lifting for DL2 and product

The proof of shadow-lifting for DL2 and product DLs provides an easy illustration of the use of the definition of the previous section (Sect. 5.1).

For DL2, the first thing to observe is that the semantics of a vector of real numbers can simply be written as an iterated sum using the notation ``_ to address elements of row-vectors:

```
Definition dl2_and {R : fieldType} {n} (v : 'rV[R]_n) := \sum_(i < n) v ``_ i.
```

Shadow-lifting for DL2 really just amounts to checking that the partial derivatives of the function $\vec{v} \mapsto \sum_{j<|\vec{v}|} \vec{v}_j$ are 1, i.e., considering vectors of size `M.+1`:

```
Lemma shadowlifting_dl2_andE (p : R) : p > 0 ->
  forall i, ('d (@dl2_and R M.+1) '/d i) (const_mx p) = 1.
```

Since the partial derivatives are all positive, DL2 satisfies the `shadow_lifting` predicate, see [24, file `dl2.v`].

Similarly, we observe for the product DL that the semantics of a vector is the function $\vec{v} \mapsto \prod_{j<|\vec{v}|} \vec{v}_j$ whose partial derivatives are p^M, which is positive:

```
Lemma shadowlifting_product_andE p : p > 0 ->
  forall i, ('d (@product_and R M.+1) '/d i) (const_mx p) = p ^+ M.
```

5.3 formalisation of L'Hôpital's rule using MathComp-Analysis

As indicated in the introduction of this section, the key technical lemma to prove shadow-lifting for STL is L'Hôpital's rule, that we show how to formalise in MathComp-Analysis. As a reminder, here follows one of L'Hôpital's rules:

▶ **Theorem 9** (L'Hôpital's rule). *Let $f, g : \mathbb{R} \to \mathbb{R}$ be functions differentiable on an open interval U except possibly at one point a. Suppose that $\forall x \in U$, $x \neq a$, we have $g'(x) \neq 0$. If it holds that $f(a) = g(a) = 0$, then if $\lim_{x \to a^+} \frac{f'(x)}{g'(x)} = l$ for some real number l, then $\lim_{x \to a^+} \frac{f(x)}{g(x)} = l$.*

It can be formally stated with MathComp-Analysis using: (a) the relation `is_derive` (between a function and its derivative at some point: the `1` appearing in the `is_derive` expression is the direction of the derivative, which is 1 for real number-valued functions) and (b) the *right filter* `a^'+` (i.e., the filter of neighborhoods of `a` intersected with $(\mathtt{a}, +\infty)$). We slightly generalize the above statement by considering a neighborhood `U` of `a` instead of an open (line 2) and by having the derivative of `g` non-zero "near" `a` (line 6) [2, Sect. 3.2]:

```
1  Context {R : realType}.
2  Variables (f df g dg : R -> R) (a : R) (U : set R) (Ua : nbhs a U).
3  Hypotheses (fdf : forall x, x \in U -> is_derive x 1 f (df x))
4             (gdg : forall x, x \in U -> is_derive x 1 g (dg x)).
5  Hypotheses (fa0 : f a = 0) (ga0 : g a = 0)
6             (cdg : \forall x \near a^', dg x != 0).
7  Lemma lhopital_right (l : R) :
8    df x / dg x @[x --> a^'+] --> l -> f x / g x @[x --> a^'+] --> l.
```

The proof is textbook, relying in particular on Cauchy's Mean Value Theorem (MVT), the proof of which can be derived from the already available MVT, see [24, file `stl.v`]. Note that we also need the variant for the left filters.

5.4 Shadow-lifting for STL

Compared with DL2 and product, the conjunction of STL (and_S in Table 1) is much more involved: it consists of two non-trivial cases (marked as $a_{\min} < 0$ and $a_{\min} > 0$ in Table 1) whose computation requires summations of exponentials of deviations. Varnai et al. provide a proof sketch for the case $a_{\min} > 0$ [27, Sect. V] which we have successfully formalised, using in particular l'Hôpital's rule from the previous section. Below we explain the formalisation of the other case $a_{\min} < 0$ that Varnai et al. did not treat.

The case $a_{\min} < 0$ actually refers to the semantics provided by the function `stl_and_lt0` already presented in Fig. 4. The positive limit we are looking for is actually $\frac{1}{M+1}$ (where $M+1$ is the size of vectors), i.e., our goal is to prove formally (the notation `\o` is for function composition):

```
Lemma shadowlifting_stl_and_lt0 (p : R) : p > 0 -> forall i,
  ('d (stl_and_lt0 \o seq_of_rV) '/d i) (const_mx p) = M.+1%:R^-1.
```

This boils down to proving the existence of the limit "from below" and "from above". The "from below" case consists in the following convergence lemma:

```
Lemma shadowlifting_stl_and_lt0_cvg_at_left (p : R) i : p > 0 ->
  h^-1 * (stl_and_lt0 (seq_of_rV (const_mx p + h *: err_vec i)) -
          stl_and_lt0 (seq_of_rV (const_mx p))) @[h --> 0^'-] --> M.+1%:R^-1.
```

For the sake of clarity, let us switch to standard mathematical notations and assume without loss of generality that `i` is actually `M`. By mere algebraic transformations (using MathComp's algebra theory), the goal can be turned into a sum of two limits:

$$\begin{aligned}
&\lim_{h\to 0^-} \frac{[\![\bigwedge_M(p,\dots,p,p+h)]\!]_{\mathrm{STL}} - [\![\bigwedge_M(p,\dots,p)]\!]_{\mathrm{STL}}}{h} \\
&= \lim_{h\to 0^-} \tfrac{1}{h}\left(\frac{(p+h)Me^{\frac{-h}{p+h}}e^{\nu\frac{-h}{p+h}} + p + h}{Me^{\nu\frac{-h}{p+h}} + 1} - p\right) && \text{by definition (see Table 1)} \\
&= \underbrace{\lim_{h\to 0^-} \frac{h}{h(M + e^{\nu\frac{h}{p+h}})}}_{(a)} + \underbrace{\lim_{h\to 0^-} \frac{M(p+h)e^{\frac{-h}{p+h}} - pM}{h(M + e^{\nu\frac{h}{p+h}})}}_{(b)} && \text{by simplification}
\end{aligned}$$

We can show directly that $(a) = \frac{1}{M+1}$ but the computation of (b) requires L'Hôpital's rule:

$$\begin{aligned}
(b) &= \lim_{h\to 0^-} \frac{\frac{hMe^{\frac{-h}{p+h}}}{p+h}}{e^{\nu\frac{h}{p+h}} + he^{\nu\frac{-h}{p+h}}(\frac{\nu}{p+h} - \frac{h\nu}{(p+h)^2}) + M} \\
&= \lim_{h\to 0^-} h \lim_{h\to 0^-} \frac{Me^{\frac{-h}{p+h}}}{p+h} \lim_{h\to 0^-} \frac{1}{e^{\nu\frac{h}{p+h}} + he^{\nu\frac{-h}{p+h}}(\frac{\nu}{p+h} - \frac{h\nu}{(p+h)^2}) + M} \\
&= 0 \cdot \frac{M}{p} \cdot \frac{1}{1+M} = 0
\end{aligned}$$

Barring the necessity of finding the most convenient breakdown of the limit in the two penultimate steps, this proof is arguably mathematically straightforward. The corresponding mechanised proof is however significantly less trivial than in the cases of product and DL2 (Sect. 5.2): length-wise the first tentative formal proof we wrote was an order of magnitude larger.

Proving the "from above" above consists of a simpler but similar argument:

$$\begin{aligned}
&\lim_{h\to 0^+} \frac{[\![\bigwedge_M(p,\dots,p,p+h)]\!]_{\mathrm{STL}} - [\![\bigwedge_M(p,\dots,p)]\!]_{\mathrm{STL}}}{h} = \lim_{h\to 0^+} \frac{1}{h}\left(\frac{pM + e^{\frac{h}{p}}e^{\frac{\nu h}{p}}}{M + e^{\frac{h}{p}}} - p\right) \\
&= \lim_{h\to 0^+} \frac{1}{M + e^{\frac{\nu h}{p}}} \lim_{h\to 0^+} e^{\frac{\nu h}{p}} \lim_{h\to 0^+} \frac{e^{\frac{h}{p}} - 1}{\frac{h}{p}} = \frac{1}{M+1} \cdot 1 \cdot 1 = \frac{1}{M+1}
\end{aligned}$$

Combined with the formalisation of the case $a_{\min} > 0$ sketched by Varnai et al. in [27, Sect. V], this completes the formal proof of shadow-lifting for STL.

Table 3 Overview of the formalisation [29].

File	Contents	L.o.c.
Additions to MathComp libraries		
`mathcomp_extra.v`	Lemmas iterated min/max, etc.	504
`analysis_extra.v`	L'Hôpital's rule, Cauchy's MVT (§ 5.3), etc.	820
Generic logic and generic definitions of properties		
`ldl.v`	LDL syntax and semantics (§ 3), shadow-lifting (§ 5.1)	417
Soundness, logical and geometric properties of concrete logics		
`dl2.v`	DL2: logical (§ 4), geometric (§ 5.2)	259
`fuzzy.v`	Gödel, Łukasiewicz, Yager, product: logical (§ 4), geometric (§ 5.2)	731
`stl.v`	STL: logical (§ 4), geometric (§ 5.4)	977
Alternative formalisations of logical properties/ soundness using extended reals		
`dl2_ereal.v`	DL2: logical (§ 4.2)	211
`stl_ereal.v`	STL: logical (§ 4.2)	362
	Total	4281

6 Conclusions, related and future work

We have presented a complete Coq formalisation of a range of existing DLs; making the following two main contributions:

1. We contribute to the DL community by *revisiting semantics of STL and DL2* in a way more amenable to formal verification. We find and fix errors in the literature.
2. We propose a *general formalisation strategy* based on dependent types and formal mathematics. The proposed formalisation is built to be easily extendable for future studies of different DLs.

Table 3 summarises the Coq implementation. Both L'Hôpital's rule and geometric properties, especially in complex cases such as STL, form a substantial part of the development. The proofs for fuzzy DLs (file `fuzzy.v`) are grouped together as thanks to their similarity, they share some of the proofs. The files `mathcomp_extra.v` and `analysis_extra.v` contain utility lemmas for the respective libraries. The file `mathcomp_extra.v` has a selection of lemmas on big operations (e.g., iterated sums, n-ary maximum), including lemmas for said operations when restricted to the domain $[0, 1]$ used by fuzzy logics. The file `analysis_extra.v` on the other hand contains proofs of L'Hôpital, Cauchy's MVT, and multiple lemmas on properties of `mine` and `maxe` (min and max for extended reals).

During our work, we completed missing parts of the shadow-lifting proof for STL, for example, the original STL proof failed to acknowledge the need for L'Hôpital's rule. We believe that understanding, let alone verifying theories that pertain to AI, without any mechanised support is difficult. Our previous attempt [24] to do this with pen and paper proofs was drowned in low-level case analysis and resulted in some errors, see Example 7. This complexity was our initial motivation to undertake the formalisation.

Regarding the concrete formalisation strategy, it was revealing that most of our formalisation was coherent with the standard MathComp libraries (and standard mathematical results), and the library extensions we needed were natural (e.g., L'Hôpital's rule). This work hence demonstrates that Coq and MathComp are effective working tools to formalise state-of-the-art AI results: DL2 and STL were published in recent conferences [16, 27] and this paper formalises the most significant results from both.

In the end, we have a uniform formalisation where all DLs are "tamed" which provides solid ground for formalisation of methods deployed in verification of neural networks.

Related Work

As this paper illustrates, neural network verification is a new field, and its nascent methods need validation and further refinement.

In terms of programming language support for neural network verification, a tool CAISAR [17], implemented as an OCaml DSL, puts emphasis on the smooth integration of a general specification language with many existing neural network solvers. However, CAISAR does not support property-guided training. The aspiration of languages like Vehicle and CAISAR is to accommodate compilation of specifications into both neural network solver and machine learning backends. For the former, there is an on-going work on certifying the neural network solver backends [12, 14].

On the side of machine-learning backends, DLs have been previously formalised in Agda as part of the Vehicle formalisation [5], but did not extend to shadow-lifting – the part that requires extensive mathematical libraries. Property-guided training certified via theorem proving was also proposed in [10].

Relevant work on formalisation of neural networks in ITPs includes: verification of neural networks in Isabelle/HOL [8] and Imandra [15], formalisation of piecewise affine activation functions in Coq [4], providing formal guarantees of the degree to which the trained neural network will generalise to new data in Coq [7], convergence of a single-layered perceptron in Coq [21]; and verification of neural archetypes in Coq [20]. The formalisation presented here does not directly formalise neural networks.

Future Work

We plan to consider other definitions of soundness, and other DLs, including STL with revised negation. We hypothesise that Definition 6 allows for generalisation (removing the condition "$p_j = p$") and this is left for future work. We saw in Sect. 4.2 that the choice of the interpretation domains has an impact on both soundness and shadow-lifting and this choice might deserve further investigation. The trade-off between idempotence, associativity and shadow-lifting that was conjectured in [27] is reminiscent of substructural logics and suggests investigating the connection. Establishing connection of this work with the logics of Lawvere quantale by Bacci et al. [6] might also provide new tools to study DLs.

Separately from the questions of scientific curiosity and mathematical elegance, there is a question of lacking programming language support for machine learning. As tools like Vehicle [13] and CAISAR [17] are being proposed to provide a more principled approach to verification of machine learning, in the long term, compilers of these new emerging languages will require certification. And this, in turn, will demand formalisation of results such as the ones we presented here. The formalisation of DLs would hence directly contribute to certified compilation of specification languages to machine learning libraries.

References

1 Reynald Affeldt, Yves Bertot, Alessandro Bruni, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, Pierre Roux, Kazuhiko Sakaguchi, Zachary Stone, Pierre-Yves Strub, and Laurent Théry. MathComp-Analysis: Mathematical Components compliant analysis library. `https://github.com/math-comp/analysis`, 2024. Since 2017. Version 1.2.0.

2 Reynald Affeldt, Cyril Cohen, and Damien Rouhling. Formalization techniques for asymptotic reasoning in classical analysis. *J. Formaliz. Reason.*, 11(1):43–76, 2018. `doi:10.6092/ISSN.1972-5787/8124`.

3 Aws Albarghouthi. *Introduction to Neural Network Verification*. verifieddeeplearning.com, 2021. arXiv:2109.10317.

4 Andrei Aleksandrov and Kim Völlinger. Formalizing piecewise affine activation functions of neural networks in Coq. In *15th International NASA Symposium on Formal Methods (NFM 2023), Houston, TX, USA, May 16–18, 2023*, volume 13903 of *Lecture Notes in Computer Science*, pages 62–78. Springer, 2023. doi:10.1007/978-3-031-33170-1_4.

5 Robert Atkey, Matthew L. Daggitt, and Wen Kokke. Vehicle formalisation, 2024. URL: https://github.com/vehicle-lang/vehicle-formalisation.

6 Giorgio Bacci, Radu Mardare, Prakash Panangaden, and Gordon D. Plotkin. Propositional logics for the Lawvere quantale. In *39th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIX), Indiana University, Bloomington, IN, USA, June 21–23, 2023*, volume 3 of *EPTICS*. EpiSciences, 2023. doi:10.46298/ENTICS.12292.

7 Alexander Bagnall and Gordon Stewart. Certifying the true error: Machine learning in Coq with verified generalization guarantees. In *The 33rd AAAI Conference on Artificial Intelligence (AAAI 2019), The 31st Innovative Applications of Artificial Intelligence Conference (IAAI 2019), The 9th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI 2019), Honolulu, Hawaii, USA, January 27–February 1, 2019*, pages 2662–2669. AAAI Press, 2019. doi:10.1609/AAAI.V33I01.33012662.

8 Achim D. Brucker and Amy Stell. Verifying feedforward neural networks for classification in Isabelle/HOL. In *25th International Symposium on Formal Methods (FM 2023), Lübeck, Germany, March 6–10, 2023*, volume 14000 of *Lecture Notes in Computer Science*, pages 427–444. Springer, 2023. doi:10.1007/978-3-031-27481-7_24.

9 Marco Casadio, Ekaterina Komendantskaya, Matthew L. Daggitt, Wen Kokke, Guy Katz, Guy Amir, and Idan Refaeli. Neural network robustness as a verification property: A principled case study. In *34th International Conference on Computer Aided Verification (CAV 2022), Haifa, Israel, August 7–10, 2022, Part I*, volume 13371 of *Lecture Notes in Computer Science*, pages 219–231. Springer, 2022. doi:10.1007/978-3-031-13185-1_11.

10 Mark Chevallier, Matthew Whyte, and Jacques D. Fleuriot. Constrained training of neural networks via theorem proving (short paper). In *4th Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis hosted by the 21st International Conference of the Italian Association for Artificial Intelligence (AIxIA 2022), Udine, Italy, November 28, 2022*, volume 3311 of *CEUR Workshop Proceedings*, pages 7–12. CEUR-WS.org, 2022. URL: https://ceur-ws.org/Vol-3311/paper2.pdf.

11 Matthew Daggitt, Wen Kokke, Ekaterina Komendantskaya, Robert Atkey, Luca Arnaboldi, Natalia Slusarz, Marco Casadio, Ben Coke, and Jeonghyeon Lee. The Vehicle tutorial: Neural network verification with Vehicle. In *6th Workshop on Formal Methods for ML-Enabled Autonomous Systems*, volume 16 of *Kalpa Publications in Computing*, pages 1–5. EasyChair, 2023. doi:10.29007/5s2x.

12 Matthew L. Daggitt, Robert Atkey, Wen Kokke, Ekaterina Komendantskaya, and Luca Arnaboldi. Compiling higher-order specifications to SMT solvers: How to deal with rejection constructively. In *12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2023), Boston, MA, USA, January 16–17, 2023*, pages 102–120. ACM, 2023. doi:10.1145/3573105.3575674.

13 Matthew L. Daggitt, Wen Kokke, Robert Atkey, Natalia Slusarz, Luca Arnaboldi, and Ekaterina Komendantskaya. Vehicle: Bridging the embedding gap in the verification of neuro-symbolic programs, 2024. arXiv:2401.06379.

14 Remi Desmartin, Omri Isac, Grant O. Passmore, Kathrin Stark, Ekaterina Komendantskaya, and Guy Katz. Towards a certified proof checker for deep neural network verification. In *33rd International Symposium Logic-Based Program Synthesis and Transformation (LOPSTR 2023), Cascais, Portugal, October 23–24, 2023, Proceedings*, volume 14330 of *Lecture Notes in Computer Science*, pages 198–209. Springer, 2023. doi:10.1007/978-3-031-45784-5_13.

15 Remi Desmartin, Grant O. Passmore, Ekaterina Komendantskaya, and Matthew L. Daggitt. CheckINN: Wide range neural network verification in Imandra. In *24th International Symposium on Principles and Practice of Declarative Programming (PPDP 2022), Tbilisi, Georgia, September 20–22, 2022*, pages 3:1–3:14. ACM, 2022. doi:10.1145/3551357.3551372.

16 Marc Fischer, Mislav Balunovic, Dana Drachsler-Cohen, Timon Gehr, Ce Zhang, and Martin T. Vechev. DL2: training and querying neural networks with logic. In *36th International Conference on Machine Learning (ICML 2019), 9–15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 1931–1941. PMLR, 2019. URL: http://proceedings.mlr.press/v97/fischer19a.html.

17 Julien Girard-Satabin, Michele Alberti, François Bobot, Zakaria Chihani, and Augustin Lemesle. CAISAR: A platform for characterizing artificial intelligence safety and robustness. In *The IJCAI-ECAI-22 Workshop on Artificial Intelligence Safety (AISafety 2022), July 24–25, 2022, Vienna, Austria*, CEUR-Workshop Proceedings, July 2022. URL: https://hal.archives-ouvertes.fr/hal-03687211.

18 Eleonora Giunchiglia, Mihaela Catalina Stoian, and Thomas Lukasiewicz. Deep learning with logical constraints. In *31st International Joint Conference on Artificial Intelligence (IJCAI-22)*, pages 5478–5485. International Joint Conferences on Artificial Intelligence Organization, July 2022. Survey Track. doi:10.24963/ijcai.2022/767.

19 Zico Kolter and Aleksander Madry. Adversarial robustness—theory and practice. NeurIPS 2018 tutorial, 2018. Available at https://adversarial-ml-tutorial.org/.

20 Elisabetta De Maria, Abdorrahim Bahrami, Thibaud L'Yvonnet, Amy P. Felty, Daniel Gaffé, Annie Ressouche, and Franck Grammont. On the use of formal methods to model and verify neuronal archetypes. *Frontiers Comput. Sci.*, 16(3):163404, 2022. doi:10.1007/S11704-020-0029-6.

21 Charlie Murphy, Patrick Gray, and Gordon Stewart. Verified perceptron convergence theorem. In *1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 43–50, 2017.

22 Kazuhiko Sakaguchi. Reflexive tactics for algebra, revisited. In *13th International Conference on Interactive Theorem Proving (ITP 2022), August 7–10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 29:1–29:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ITP.2022.29.

23 Kazuhiko Sakaguchi and Pierre Roux. Algebra tactics: Ring, field, lra, nra, and psatz tactics for Mathematical Components. https://github.com/math-comp/algebra-tactics, 2021. Last stable release: 1.2.3 (2024).

24 Natalia Slusarz, Ekaterina Komendantskaya, Matthew L. Daggitt, Robert J. Stewart, and Kathrin Stark. Logic of differentiable logics: Towards a uniform semantics of DL. In *24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2023), Manizales, Colombia, June 4–9, 2023*, volume 94 of *EPiC Series in Computing*, pages 473–493. EasyChair, 2023. doi:10.29007/C1NT.

25 Mathematical Components Team. Mathematical components library, 2007. Last stable version: 2.2.0 (2024). URL: https://github.com/math-comp/math-comp.

26 Emile van Krieken, Erman Acar, and Frank van Harmelen. Analyzing differentiable fuzzy logic operators. *Artif. Intell.*, 302:103602, 2022. doi:10.1016/J.ARTINT.2021.103602.

27 Péter Várnai and Dimos V. Dimarogonas. On robustness metrics for learning STL tasks. In *2020 American Control Conference (ACC 2020), Denver, CO, USA, July 1–3, 2020*, pages 5394–5399. IEEE, 2020. doi:10.23919/ACC45564.2020.9147692.

28 J Łukasiewicz. *O logice trójwartościowej (in Polish). English translation: On Three-Valued Logic, in Borkowski, L.(ed.) 1970. Jan Łukasiewicz: Selected Works, Amsterdam: North Holland.* Ruch Filozoficzny, 1920.

29 Natalia Ślusarz, Reynald Affeldt, and Alessandro Bruni. Formalisation of Differentiable Logics in Coq, 2024. Software, swhId: swh:1:dir:bd213b761dfc453ccfe8e785a38cffe583c98f04 (visited on 2024-08-21). URL: https://github.com/ndslusarz/formal_LDL.