Lecture Notes on
Truth is Ephemeral

15-836: Substructural Logics
Frank Pfenning

Lecture 1
August 29, 2023

1 Introduction

When studying logic in an introductory course we are used to thinking of truth as a mathematical concept, something that is objective and unalterable like the laws of physics. Truth is something we may be able to uncover and understand, but not something we can create. In this course I will try to convince you otherwise.

First, truth is ephemeral. At this point in lecture I held a piece of chalk. Direct evidence confirmed: "*Frank holds a piece of chalk.*" After I put it down on the table in front of me this proposition was no longer true. Hence, evidently, *truth is ephemeral*. Substructural logics capture and analyze this phenomenon, which is of fundamental importance in computer science. For example, while executing a program in an imperative language a variable x may hold the value 5. That's an ephemeral truth, because after assigning x the value 7 it is no longer true—instead, the value of x is then 7.

Second, certain truths are persistent. For example, given the plethora of proofs, it is difficult to deny the Pythagorean Theorem. Or, by the very nature of implication, A always implies A for any proposition A. Substructural logics account for ephemeral as well as persistent truths and therefore *generalize* rather than replace "traditional" logics that study persistent truth. In the context of this course we call such logics *structural*. The origin of the terms *structural* and *substructural* will become clear in the course of this lecture.

Logic is the study of the laws of valid inference, so we start the course in the same way. In today's lecture we avoid logical connectives entirely, just using rules of inference. It turns out that we can use inference rules to describe some algorithms that can be seen as performing logical inferences, which is one of the many connections between logic and computer science.

2 Structural Inference

Consider a relation $\mathsf{edge}(x, y)$ between vertices x and y in a directed graph. We would like to define when there is a path from x to y. Mathematically, we might say that the path relation is the transitive closure of the edge relation. We define this with two rules of inference:

$$\frac{\mathsf{edge}(x, y)}{\mathsf{path}(x, y)}\ \mathsf{Edge} \qquad \frac{\mathsf{path}(x, y) \quad \mathsf{path}(y, z)}{\mathsf{path}(x, z)}\ \mathsf{Trans}$$

Some terminology: the propositions above the line in a rule of inference are called *premises*, the propositions below *conclusions*. The variables in a rule (here x, y, and z) are called *schematic variables*.

The process of inference starts from a given state of knowledge and deduces additional propositions that must be true, according to the rules of inference. We say we *apply a rule of inference*, given a particular instantiation of the schematic variables.

Here is a small example: our initial state of knowledge is $\mathsf{edge}(a, b)$, $\mathsf{edge}(b, c)$, $\mathsf{edge}(b, d)$ for some vertices a, b, c, and d. We apply all possible inferences at each stage, but if a conclusion is already in our database of facts we don't write it down again. The justifications are just the inference rules applied to the labels of all the premises.

(1)	$\mathsf{edge}(a, b)$	given
(2)	$\mathsf{edge}(b, c)$	given
(3)	$\mathsf{edge}(b, d)$	given
(4)	$\mathsf{path}(a, b)$	$\mathsf{Edge}(1)$
(5)	$\mathsf{path}(b, c)$	$\mathsf{Edge}(2)$
(6)	$\mathsf{path}(b, d)$	$\mathsf{Edge}(3)$
(7)	$\mathsf{path}(a, c)$	$\mathsf{Trans}(4, 5)$
(8)	$\mathsf{path}(a, d)$	$\mathsf{Trans}(4, 6)$

At the last stage we have reached *saturation*: any way we can apply inference rules will result in conclusions we already know. Because we think of the rules as *defining* the propositions involved (specifically $\mathsf{path}(x, y)$ in this example) we stop inference at this point. If we want to know if there is a path we can just look it up on this final, saturated state. For example, there is a path from a to d, but there is no path from b to a.

A few remarks about the process of inference. In general, it will not be the case that given some initial facts we reach saturation. For example, rule that allows us to conclude that $\mathsf{nat}(\mathsf{s}(x))$ if $\mathsf{nat}(x)$ would lead to an infinite set of facts if 0 is also known to be a natural number. In our example, the size of the database is bounded by $2n^2$ where n is the number of vertices.

Secondly, the justification gives us a proof of each derived fact. For example, the proof of $\mathsf{path}(\mathsf{a}, \mathsf{d})$ would be

$$\dfrac{\dfrac{\mathsf{edge}(a,b)}{\mathsf{path}(a,b)}\,\mathsf{Edge} \quad \dfrac{\mathsf{edge}(b,d)}{\mathsf{path}(b,d)}\,\mathsf{Edge}}{\mathsf{path}(a,d)}\,\mathsf{Trans}$$

Such a proof can also be expressed as a term, thinking of the inference rule as a term constructor. In this example, this might be written as

$$\mathsf{Trans}(\mathsf{Edge}(1), \mathsf{Edge}(3))$$

substituting the proof terms for (4) and (6).

In general, a proposition may have multiple different proofs, including infinitely many. For example, if we add an edge from b to a, then we can cycle from a to a as many times as we wish. It is therefore important that we do not take the proofs into account when we decide saturation: we want a finite number of facts (each with at least one proof) but not necessarily a finite number of proofs.

We also observe that the order in which we perform inferences is nondeterministic but entirely irrelevant: the saturated state will always be the same. This is an example of so-called *don't-care nondeterminism*. If we tracked proofs, though, they could be different based on the order in which we performed the inferences.

Because (a) the order of the propositions in a state does not matter and (b) knowing a fact once is just as good as "knowing it twice", states form a *set*. Expressed as two laws: $P, Q = Q, P$ and $P, P = P$. Because we view this as an *equality* between states, it can be applied anywhere in a state. We call these properties *exchange* and *contraction*, respectively, although the latter may be called *idempotence* in algebra.

Datalog (see, for example, Maier et al. [2018] or Green et al. [2012]) is a programming language based on structural inference with some additional features such as stratified negation and constraints. Applications in computer science include *program analysis* [Whaley et al., 2005, Smaragdakis and Bravenboer, 2010] and algorithms such as subtyping [DeYoung et al., 2023]. An interesting sidebar is that there are some *meta-complexity theorems* that allows us to read off the complexity of algorithms from the inference rules that define the algorithms [Ganzinger and McAllester, 2001, 2002].

Because it is difficult to express many common algorithms and programming idioms just by inference, we see structural inference primarily as a way to express algorithms that can then be implemented as a library in a language with a more complete set of constructs. Our small and unrealistic implementation as part of Assignment 1 is a tiny example of such a library.

3 Linear Inference

Technically, linear inference arises from structural inference by denying *contraction* while keeping exchange. This means that the state becomes a *multiset* (or *bag*) rather than being a set. Even more importantly, when applying an inference rule we *remove* the premises from a state and then add in the conclusions.

As a first example we consider *coin exchange*: a quarter can be exchanged for two dimes and a nickel, and a dime can be exchanged for two nickels. We can also do the reverse exchange.

$$\frac{q}{d \quad d \quad n}\ \mathsf{Q} \qquad \frac{d \quad d \quad n}{q}\ \overline{\mathsf{Q}} \qquad \frac{d}{n \quad n}\ \mathsf{D} \qquad \frac{n \quad n}{d}\ \overline{\mathsf{D}}$$

The first and third rules exemplify something new, namely rules with multiple conclusions. We cannot replace them with multiple rules, each with a single conclusion since the premises will be consumed from the state during rule application.

Linear inference represents a change of state. Below we show the whole state and all results of possible inferences. As an example, in the first step we replace q by d, d, n, applying the rule Q. The additional n is just carried over.

(1)	q, n	given
(2)	d, d, n, n	$\mathsf{Q}(1)$
(3)	d, n, n, n, n	$\mathsf{D}(2)$
(4)	d, d, d	$\overline{\mathsf{D}}(2)$
(5)	n, n, n, n, n, n	$\overline{\mathsf{D}}(3)$

Note that the proof notation is somewhat approximate now: it only says which rule is applied to which line, but the premises of a rule match only a portion of the state while carrying over the remainder.

At the end of this process the application of any rule to any state will result only in a state already in our collection. We have saturated the set of possible states, rather than arrived at a single saturated state. It should be clear that termination must be redefined in this manner because inference does not monotonically increase our knowledge. Furthermore, just as for structural inference, termination is not guaranteed in general. For example, if we added a rule that allowed us to conclude d, d from d we could produce arbitrarily many dimes starting from any state with at least one.

We accomplish the change of state at the technical level by denying the structural rule of contraction, but we keep the rule of exchange. Linear inference is therefore an example of *substructural inference*.

At first one might think that the right way to handle the "holding-the-chalk" example from the introduction by using a *temporal logic*. That is, at some time t I am holding the chalk, and at some time $t + \delta$ I no longer hold the chalk. And, indeed,

temporal logic has a significant role both in philosophy and computer science. For capturing a change of state (like putting down a piece of chalk, or assigning to a variable) it has a severe drawback: it suffers from the *frame problem*. When I put down my piece of chalk in lecture, all of you remained in your seats. When we assign to a single variable, all the other variables in the program remain unchanged. So we would also have to say "*nothing else changes*", but it is difficult to circumscribe the meaning of "*nothing else*". It depends, and is therefore inherently anti-modular. If we add another variable to our program, for example, suddenly "*nothing else*" would have to include the new variable, even if in some sense it has nothing to do with our particular assignment.

Substructural inference provides an intrinsic solution to the frame problem since the premises of a rule are always matched against a portion of the state. Inherently, no matter what else we add, only the matched portion of the state changes and the remainder is carried over. Therefore, a priori, substructural logic is the better choice for a controlled change of state. Temporal logic is suitable when, intrinsically, many things happen in parallel (e.g., in a logic circuit) or when we want to reason about temporal aspects of a computation that is described by state change. Linear inference has its roots in *linear logic* [Girard, 1987], although the state-changing aspect was formulated perhaps most explicitly in *multiset rewriting* [Cervesato et al., 2000, Cervesato, 2000, Cervesato and Scedrov, 2009] and extended in the Concurrent Logical Framework (CLF) [Watkins et al., 2002, Cervesato et al., 2002, Schack-Nielsen and Schürmann, 2008, Schack-Nielsen, 2011].

4 Ordered Inference

We can go one step further in the exploration of substructural inference by also removing exchange. This means the state is now just a sequence of propositions. We write ordered states without a comma to remind ourselves of the lack of exchange. Algebraically, the state is a monoid, the unit element being the empty state.

The example we start with is recognizing a word consisting of matching parentheses. The constituents of the state are left and right parentheses. We have just one rule of inference

$$\frac{(\quad)}{\cdot}\ \textsf{Cancel}$$

This is an inference rule with zero conclusions, something that would be entirely pointless in structural inference but makes sense for substructural inference. We run through an example, but we don't bother showing the inference rule that was

applied because there is only one.

(1)	(	)	(	(	)	)	(	)	given
(2)			(	(	)	)	(	)	from (1)
(3)	(	)	(			)	(	)	from (1)
(4)	(	)	(	(	)	)			from (1)
(5)			(			)	(	)	from (2), (3) [in 3 ways], or (4)
(6)			(	(	)	)			from (2) or (4)
(7)							(	)	from (5) [in 2 ways] or (6)
(8)									from (7)

Despite the suggestive way we have written the states to make them easier to relate, note that the exact state (5) (namely, ( ) ( )) can be deduced in 5 different ways but we only list it once.

We call the final state (8) *quiescent* because no inference rule can be applied to it.

The claim is that given an ordered state consisting of left and right parentheses, we can deduce the empty state if and only if the parentheses match in the given order. In particular, if we start from a state where the parentheses don't match, we cannot reach the empty state. For example:

(1)	)	(	)	(	given
(2)	)			(	from (1)

Here, we are stuck after one inference can cannot deduce anything new. In other words, the final state (2) is quiescent but not empty.

We observe that in this example we don't actually need to explore *all* reachable states. We can arbitrarily apply cancellation (if possible) and then continue from the resulting state until we reach a quiescent state. We call this *don't-care nondeterminism*: the rules could be applied in multiple ways, but it is correct to pick an arbitrary one. This is sufficient to determine if the initial state represents matching parentheses.

The rules themselves don't contain the assumptions about the initial state, the way the rules are to be applied (in a don't-care nondeterministic manner until quiescent or generating all reachable states), or how to interpret the final state(s). So we should always give this information explicitly.

1. We are given an initial ordered state consisting of left '(' and right ')' parentheses.

2. We apply cancellation in a don't-care nondeterministic manner until we reach a quiescent state.

3. The quiescent state is empty if and only if the initial state had matching parentheses.

We say the problem representation by (structural, linear, or ordered) inference is *adequate* if we can prove, at the meta-level the third part, given the circumstances of the first and second parts.

Here is what we might say for the coin exchange:

1. We are given an initial linear state consisting of quarters ('q'), dimes ('d') and nickels ('n').
2. We apply the rules in *don't-know nondeterministic* manner until we have deduced all reachable states.
3. A state is reachable if and only if it has the same total monetary value as the initial state and consists only of quarters, dimes, and nickels.

5 Binary Increment as Ordered Inference

We explore incrementing a binary number as a second example of ordered inference. We have propositions 0 (bit 0), 1 (bit 1), and ϵ (end of number, not to be confused with the empty word) to represent a natural number in binary form. For example, the number 6 would be represented as the state ϵ 1 1 0. Furthermore, we have the proposition inc which is meant to increment the binary number to its left. We capture this meaning with the following rules:

$$\frac{0 \quad \mathsf{inc}}{1}\ \mathsf{inc}_0 \qquad \frac{1 \quad \mathsf{inc}}{\mathsf{inc} \quad 0}\ \mathsf{inc}_1 \qquad \frac{\epsilon \quad \mathsf{inc}}{\epsilon \quad 1}\ \mathsf{inc}_\epsilon$$

In the inc_1 rule, the increment in the conclusion represents the carry. We run a small example, incrementing the number 5 by 2 to obtain 7. Each line is inferred from the preceding one.

(1)	ϵ	1	0	1	inc	inc	given
(2)	ϵ	1	0	inc	0	inc	by inc_1
(3)	ϵ	1	1		0	inc	by inc_0
(4)	ϵ	1	1		1		by inc_0

In state (2) we had a choice between applying inc_0 to the first or second occurrence of inc, and we arbitrarily picked the first one. Either way would have led to the same quiescent state at the end. We express adequacy using regular expression notation to capture the permissible forms of the ordered state.

1. We are given an initial state in the form $\epsilon\ (0 \mid 1)^*\ \mathsf{inc}^*$.
2. We apply rules in a don't-care nondeterministic manner until we reach quiescence. Each intermediate state will have the form $\epsilon\ (0 \mid 1 \mid \mathsf{inc})^*$.
3. We have computed the output in the form of a final state $\epsilon\ (0 \mid 1)^*$ when we have reached quiescence.

6 Blocks World as Linear Inference

As a final example in today's lecture we present a version of the classic blocks world planning problem as linear inference. We have a robot hand that can hold a single block and a table with possible severals stacks of labeled blocks. The robot hand can pick up a block from the top of a stack and put it down on top of another stack or an empty spot on the table. We can either explore all reachable states, or create a plan to reach a particular goal state.

We start with the following propositions.

- empty (the robot hand is empty)
- $\mathsf{holds}(x)$ (the robot hand holds block x)
- $\mathsf{on}(x, y)$ (block x is on top of block y)

A plausible first attempt at a rule for picking up a block would be to state that we can pick up a block x if (a) the hand is empty, and (b) is no other block on top of it. In addition, if x is on some other block y then after we pick up x it is no longer on y so we need to remove that fact from the state.

$$\frac{\mathsf{empty} \quad \neg\exists z.\, \mathsf{on}(z, x) \quad \mathsf{on}(x, y)}{\mathsf{holds}(x)}\ \mathsf{pickup?}$$

The difficulty with this rule is that the middle premise is intended to check a condition on the state without actually altering the state. For one, we don't have the means to express this since we didn't want to use logical connectives (like negation and the existential quantifier) in this lecture. Perhaps even more troubling is that it would violate our fundamental definition of linear inference which allows us to match the premises against an *arbitrary* portion of the state and carry over the remainder unchanged. But if have a state such as $\mathsf{empty}, \mathsf{on}(b, \mathsf{table}), \mathsf{on}(a, b)$ then adding $\mathsf{on}(c, a)$ would suddenly render the rule inapplicable.

The solution is to express the condition that allows us to pick up a block in a positive way, as another proposition. We then need to maintain the new proposition as part of the inference rules. Here, we add $\mathsf{clear}(x)$ to express that block x is clear, that is, there is nothing on top of x. This should be true exactly if there does not exist a z such that z is on x. The rule then becomes

$$\frac{\mathsf{empty} \quad \mathsf{clear}(x) \quad \mathsf{on}(x, y)}{\mathsf{holds}(x) \quad \mathsf{clear}(y)}\ \mathsf{pickup}$$

Notice that application of this rule will remove $\mathsf{clear}(x)$ from the state (the hand now holds it) and adds $\mathsf{clear}(y)$ (x is no longer on top of y).

We can think of the table itself as a pseudo-block, and mark the empty slots on the table as clear. For example, if there are two spots on the table and on one we have the stack a on top of b we would represent this as the state

$$\mathsf{empty}, \mathsf{clear}(a), \mathsf{on}(a, b), \mathsf{on}(b, \mathsf{table}), \mathsf{clear}(\mathsf{table})$$

In this state we can only pick up a, which would get us to the state

$$\mathsf{holds}(a), \mathsf{clear}(b), \mathsf{on}(b, \mathsf{table}), \mathsf{clear}(\mathsf{table})$$

We can not pick up the table (the pseudo-block) because there is no proposition $\mathsf{on}(\mathsf{table}, x)$. Putting down a block is just the inverse of the rule for picking one up.

$$\frac{\mathsf{holds}(x) \quad \mathsf{clear}(y)}{\mathsf{empty} \quad \mathsf{clear}(x) \quad \mathsf{on}(x, y)}\ \mathsf{putdown}$$

At this point it should be clear that we can easily infer all the reachable state from a valid initial state. But what is a valid initial state? This is not so easy to characterize. For example, we don't want to allow "circular stacks" with $\mathsf{on}(x, y)$ and also $\mathsf{on}(y, x)$. We don't want to allow the table to be on anything. We don't want to allow two different blocks to have the same label a. We don't want the hand to be empty and hold a block at the same time, because it would mean we really have two hands. A general technique for describing valid linear states (including valid initial states, which is the same in this case) is via *generative grammars* [Simmons, 2012], which are beyond the scope of the present lecture, but which we may return to in a future lecture. Here we contend ourselves by saying that the state should consist of distinct blocks and should be "physically possible".

1. We are given a linear initial state consisting of distinct blocks and a finite number of empty spots on the table $\mathsf{clear}(\mathsf{table})$ in a physically possible configuration.
2. Inference rules are applied in a don't-know nondeterministic way.
3. A state is reachable by robot actions from the initial state if and only if we can reach it via linear inference.

Similar to the example of graph reachability, the actual plan for achieving a goal state is encoded in its proof. We will return to the notion of proof at the beginning of the next lecture.

7 Summary

We have introduced three forms of logical inference, without using any notion of logical connective.

Structural inference. States are sets of propositions representing the current state of knowledge that grows monotonically during inference. A state is *saturated* if any inference only deduces facts we already know. Some algorithms can be expressed as sets of inferences rules that must saturate, applying rules in a don't-care nondeterministic manner. It is called *structural* because states are identified up to the structural rules of *exchange* and *contraction*.

Linear inference. States are multisets of propositions. Linear inference consumes the premises of a rule and adds its conclusions, thereby describing a change of state. We can perform don't-know nondeterministic inferences until we have deduced all reachable states, or we can perform don't-care nondeterministic inference until we reach *quiescence* where no further rules can be applied. Linear inference is a form of *substructural inference* because we deny the law of contraction for states (while keeping the law of exchange).

Ordered inference. States are sequences of propositions. Inferences apply to consecutive propositions, replacing them with the (also consecutive) conclusions. Like for linear inference, we can explore all possible reachable states or proceed in a don't-care nondeterministic manner to reach a quiescent state. Ordered inference is another form of substructural inference, denying both the laws of exchange and contraction.

References

Iliano Cervesato. Typed multiset rewriting specifications of security protocols. In A. Seda, editor, *Proceedings of the First Irish Conference on the Mathematical Foundations of Computer Science and Information Technology (MFCSIT'00)*, Cork, Ireland, July 2000. Elsevier Electronic Notes in Theoretical Computer Science. To appear.

Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207(10):1044–1077, October 2009.

Iliano Cervesato, Nancy A. Durgin, Max Kanovich, and Andre Scedrov. Interpreting strands in linear logic. In E. Clarke, N. Heintze, and H. Veith, editors, *Proceedings of the Workshop on Formal Methods and Computer Security (FMCS'00)*, Chicago, Illinois, July 2000.

Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

Henry DeYoung, Andreia Mordido, Frank Pfenning, and Ankush Das. Parametric subtyping for structural parametric polymorphism. *CoRR*, abs/2307.13661, July 2023. URL https://arxiv.org/abs/2307.13661. Submitted.

Harald Ganzinger and David A. McAllester. A new meta-complexity theorem for bottom-up logic programs. In T. Nipkow R. Goré, A. Leitsch, editor, *Proceedings of the First International Joint Conference on ArAutomated Reasoning (IJCAR'01)*, pages 514–528, Siena, Italy, June 2001. Springer-Verlag LNCS 2083.

Harald Ganzinger and David A. McAllester. Logical algorithms. In P. Stuckey, editor, *Proceedings of the 18th International Conference on Logic Programming*, pages 209–223, Copenhagen, Denmark, July 2002. Springer-Verlag LNCS 2401.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5(2):105–195, 2012.

David Maier, K. Tuncay Tekle, Michael Kifer, and David S. Warren. Datalog: Concepts, history, and outlook. In Michael Kifer and Yanhong Annie Liu, editors, *Declarative Logic Programming: Theory, Systems, and Applications*, pages 3–100. ACM and Morgan & Claypool, 2018.

Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen, January 2011.

Anders Schack-Nielsen and Carsten Schürmann. Celf - a logical framework for deductive and concurrent systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR'08)*, pages 320–326, Sydney, Australia, August 2008. Springer LNCS 5195.

Robert J. Simmons. *Substructural Logical Specifications*. PhD thesis, Carnegie Mellon University, November 2012. Available as Technical Report CMU-CS-12-142.

Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In O. de Moor, G. Gottlob, T. Furche, and A. Sellers, editors, *Datalog Reloaded*, pages 245–251, Oxford, UK, March 2010. Springer LNCS 6702. Revised selected papers.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog and binary decision diagrams for program analysis. In K.Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*, pages 97–118. Springer LNCS 3780, November 2005.