Talk Notes: Substructural Logics

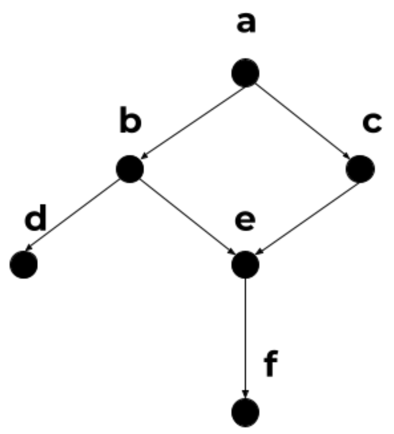
Programming Research Laboratory (PRL) Seminar
Northeastern University

Chris Martens[*]

September 2023

1 Computing with Inference Rules

Consider the problem of deriving all reachable nodes in a directed graph from a single starting point, e.g. in the following graph:



We can describe this problem as a deductive system. First, we represent the graph as a collection of edge predicates:

$$\mathsf{edge}(a,b), \mathsf{edge}(a,c), \mathsf{edge}(b,d),$$
$$\mathsf{edge}(b,e), \mathsf{edge}(c,e), \mathsf{edge}(e,f)$$

Then we define inference rules that define a path relation as the reflexive, transitive closure over edges.

[*]With gratitude to Frank Pfenning for making his course notes, on which this note is heavily based, publicly available.

$$\frac{\mathsf{edge}(X,Y)}{\mathsf{path}(X,Y)}\ \mathsf{path/edge} \qquad \frac{\mathsf{edge}(X,Y) \quad \mathsf{path}(Y,Z)}{\mathsf{path}(X,Z)}\ \mathsf{path/trans}$$

This notation means: if we know all of the facts that sit above the horizontal line (the *premises*), then we are allowed to deduce the one beneath it (the *conclusion*). The label sitting off to the side is simply a name we have chosen for the rule.

By themselves, these inference rules have no "operational" definition, and therefore do not constitute an algorithm for solving the problem. They are simply a mathematical representation of what it *means* to be a path, stated in the language of proof theory (inference rules) rather than in the language of set theory (for example).

However, we can also imaging "running" these rules. Suppose we think of our collection of edge predicates as an initial state—the *state* here is the set of facts we know. We can transition states by *applying* any applicable rules of inference. So for example, we'd first collect all the facts from applying the path/edge rule to each of the edges:

$$\mathsf{path}(a,b), \mathsf{path}(a,c), \mathsf{path}(b,d),$$
$$\mathsf{path}(b,e), \mathsf{path}(c,e), \mathsf{path}(e,f)$$

From there, we can compute all two-step paths by applying the path/trans rule (to our original edge facts along with these newly derived facts):

$$\mathsf{path}(a,d), \mathsf{path}(a,e), \mathsf{path}(b,d), \mathsf{path}(b,f), \mathsf{path}(c,f)$$

(On the board, I will draw the explicit justifications for each new derivation.)

Note that, although we were able to derive $\mathsf{path}(a,e)$ in two different ways, we only added it to our collection of facts once. Intuitively, this is because once we learn something is true, we don't gain any new information from learning it again.

Finally, we can derive the three-step path

$$\mathsf{path}(a,f)$$

Note that at each step, our "knowledge state" implicitly contains every previous fact we have collected along the way. These facts can allow us to derive new ones, and we might use them to derive multiple new facts, or we might not touch them again at all.

So how do we know we can "stop" after the above derivations?

If we consider every possible way that our inference rules might apply, they would only generate knowledge that we already have. This condition is called "saturation."

Implementing this form of reasoning with inference rules is known as *forward-chaining proof search*, and it is very close to what you find in the logic programming Datalog, a close cousin of prolog.

2 Substructural Inference

Now let's think about a couple of different kinds of inference. One thing that the form *structural inference* we've been working with does not let us do is represent *state change*. Once we know something, we know it forever. But truth is often ephemeral! For example, if I have a dollar in my bank account today, there is no guarantee that I will tomorrow.

The next kind of inference we'll examine can be thought of as a kind of *resource exchange*. We'll now write our inference rules with multiple conclusions, and they should be read as: "if we have all of the premises, we can *spend* them to receive the conclusions."

2.1 Linear Inference

Here is an example about harvesting and crafting with natural resources. Perhaps we could call it "Craftmine".

$$\frac{\mathsf{tree}}{\mathsf{wood\ wood\ wood\ wood}}\ \mathsf{chopT} \qquad \frac{\mathsf{wood}}{\mathsf{plank\ plank}}\ \mathsf{chopW} \qquad \frac{\mathsf{plank}}{\mathsf{stick\ stick}}\ \mathsf{chopP}$$

$$\frac{\mathsf{stick\ plank\ plank}}{\mathsf{pickaxe}}\ \mathsf{craftP} \qquad \frac{\mathsf{stone\ pickaxe}}{\mathsf{cobble\ pickaxe}}\ \mathsf{mineS}$$

A question we might ask about this system of rules is: if I start out with one tree and three stones, can I produce three cobbles? See if you can use your intuition to sketch a forward-chaining proof that you can, using these rules.

(Not written here: the *failure* version of the derivation that gets stuck because we spend all of our planks by eagerly decomposing things.)

The sequence of states to successfully do this might look like (where we use a^n to represent the proposition a present n times):

$$\begin{array}{ll}
 & \mathsf{tree\ stone}^3 \\
\mathsf{chopT}: & \mathsf{wood}^4\ \mathsf{stone}^3 \\
\mathsf{chopW}: & \mathsf{plank}^2\ \mathsf{wood}^3\ \mathsf{stone}^3 \\
\mathsf{chopW}: & \mathsf{plank}^4\ \mathsf{wood}^2\ \mathsf{stone}^3 \\
\mathsf{chopP}: & \mathsf{plank}^3\ \mathsf{stick}^2\ \mathsf{wood}^2\ \mathsf{stone}^3 \\
\mathsf{craftP}: & \mathsf{pickaxe\ plank}^2\ \mathsf{stick\ wood}^2\ \mathsf{stone}^3 \\
\mathsf{mineS}: & \mathsf{pickaxe\ cobble\ stick\ wood}^2\ \mathsf{stone}^2 \\
\mathsf{mineS}: & \mathsf{pickaxe\ cobble}^2\ \mathsf{stick\ wood}^2\ \mathsf{stone} \\
\mathsf{mineS}: & \mathsf{pickaxe\ cobble}^3\ \mathsf{stick\ wood}^2
\end{array}$$

Or alternatively we can draw this as a DAG, to avoid repeating elements of the state that aren't involved in a given rule:

```
tree                                        stone
-----------------------------------             |
|                \           \       \          |
wood            wood        wood    wood        |
--------        ------                         /
|       \       |     \                       /
plank   plank   plank plank                  /
------     \       |                        /
|     \     \      |                       /
stick stick  \     |                      /
           \  \   /                      /
            -------                     /
            pickaxe                    /
            ---------------------------
                      |        |
                    cobble  pickaxe
```

For this kind of inference, states (contexts) are *multisets*, and state transition can *delete* elements as well as add them. But it doesn't matter what order they are in (other than to be able to draw a nicer proof DAG). Reasoning with this form of inference leads to *linear logic* [1] (or more properly *affine logic*, which drops contraction but not weakening—i.e. we don't care if there are extra premises lying around, but we do care that they are used *at most* once).

2.2 Ordered Inference

Finally, we visit a kind of inference where we actually treat proof states as ordered lists, and we don't in general allow them to commute around each other. One example, in fact the original application of this logic, is parsing natural-language expressions:

$$\frac{\mathsf{NounPhrase}\quad\mathsf{VerbPhrase}}{\mathsf{Sentence}}\ sent/nv \qquad \frac{\text{if }\mathsf{Sentence},\mathsf{Sentence}}{\mathsf{Sentence}}\ sent/if$$

$$\frac{\text{the cat}}{\mathsf{NounPhrase}}\ np/cat \qquad \frac{\text{sits}}{\mathsf{VerbPhrase}}\ vp/sit \qquad \frac{\text{fits}}{\mathsf{VerbPhrase}}\ vp/fit$$

Now a "proof" corresponds to parsing a sentence! For example, the following forward-chaining proof is valid:

$$\begin{array}{rl}
 & \text{if the cat fits, the cat sits} \\
np/cat & \text{if } \mathsf{NounPhrase} \text{ fits, the cat sits} \\
np/cat & \text{if } \mathsf{NounPhrase} \text{ fits, } \mathsf{NounPhrase} \text{ sits} \\
vp/fit & \text{if } \mathsf{NounPhrase\ VerbPhrase}\text{, } \mathsf{NounPhrase} \text{ sits} \\
vp/sit & \text{if } \mathsf{NounPhrase\ VerbPhrase}\text{, } \mathsf{NounPhrase\ VerbPhrase} \\
sent/nv & \text{if } \mathsf{Sentence}\text{, } \mathsf{NounPhrase\ VerbPhrase} \\
sent/nv & \text{if } \mathsf{Sentence}\text{, } \mathsf{Sentence} \\
sent/if & \mathsf{Sentence}
\end{array}$$

If we write this proof in tree form, we actually get grammatical parse trees, which you may have been forced to write in an English (or other language) classroom as a child.

$$\dfrac{\dfrac{\vdots}{\mathsf{NounPhrase}} \quad \dfrac{\vdots}{\mathsf{VerbPhrase}}}{\mathsf{Sentence}}$$

There are ways to make this work better for some of the irregularities of English, e.g. subject/verb agreement, which are shown in the original paper by Lambek introducing this idea [3]. They involve modeling parts of speech (language categories) as ordered implication, e.g. "if I find this kind of structure to my left, I can become this." See Lambek's paper for more details.

Exercise: What would happen if you turned all of the rules upside down? What would proofs correspond to?

Exercise: Another cool application of ordered logic is using it to rewrite grid-states, e.g. for a grid-based game. Could you write the rules for how Chess pieces move using ordered logic? There is a cool game-prototyping tool called Puzzlescript where you actually program the logic of tile-based puzzle games with a syntax very close to this.

3 Logical Connectives

If we want to reason about all the kinds of rules (like path and edge) that one might write, in any domain, rather than designing a new set of inference rules for each domain, it's useful to give some notation for representing them *within* the logic, i.e. as a class of *propositions*. This is where we get the idea of *logical connectives*: we turn our notation for representing things with inference rules into *propositional objects* that can then themselves be reasoned over within another, more general set of inference rules. We'd like to be able to describe *in general* when it's possible to derive a conclusion from some set of assumptions, i.e. when we can create a forward-inference chain of the form:

$$\begin{array}{c} x_1{:}A_1 \ldots x_n{:}A_n \\ \vdots \\ B \end{array}$$

To do this, it's helpful to use the metavariable Γ to refer to assumptions $x_1{:}A_1 \ldots x_n{:}A_n$, and to write the entire thing $\Gamma \vdash B$, read "Γ entails B". This structure—a context of assumptions, a conclusion, and the entailment relation between them—is known as a *sequent* in logic. You might have seen this kind of thing before in programming languages textbooks or papers, in the form of typing rules, like $\Gamma \vdash e : \tau$ (expression e has type τ under context Γ, where Γ contains variables with typing assumptions that are in scope when typechecking e).

To look at how inference works with sequents, let us consider our previous path example. Our "knowledge state" now lives on the left hand side of the sequent, so we will read it "upside-down", i.e. from bottom to top. For example, the path/trans rule becomes:

$$\frac{\Gamma, \mathsf{edge}(X,Y), \mathsf{path}(Y,Z), \mathsf{path}(X,Z) \vdash C}{\Gamma, \mathsf{edge}(X,Y), \mathsf{path}(Y,Z) \vdash C}$$

C here stands for some arbitrary goal we are working towards. If we read the rule bottom-up, we can interpret the conclusion as a *goal*, and the premise(s) to a rule as *subgoals*. Note that we *maintain* the facts from the bottom as we move upward!

On the right hand side of the sequent, however, we can only write one proposition.[1] So we need a logical connective to *internalize* the idea that our goals can be made of multiple things, and this connective will be conjunction ($\wedge$). We can write a general rule to represent this:

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

Again reading this rule bottom-up, it says that if our goal is to prove we can reach a knowledge state $A \wedge B$, then our subgoals are to prove we can reach A and we can reach B.

We can likewise describe how to reason, in general, if our goal is to prove a *hypothetical* derivation, which we will write as implication $A \Rightarrow B$:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$$

In our subgoal, we add A to the context and continue trying to prove B.

Now that we can shift propositions from the right side of the sequent to the left, however, we need to be able to state what we can do with these connectives on the left.

It turns out that $\wedge$ just "internalizes" our notion of state composition (i.e. ","), so we can re-interpret it that way on the left:

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C}$$

To use implication on the left, we need to be able to derive its left-hand side, after which we can continue by using its right-hand side.

$$\frac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \Rightarrow B \vdash C}$$

Using these general rules, we can prove more general kinds of facts about inference, such as $A \wedge B \Rightarrow B \wedge A$. We can also re-interpret our inference rules for specific domains as logical propositions, i.e.:

[1] If we try to do this development with multiple propositions on the right, we run into trouble when we try to define implication, i.e. reason about hypothetical judgments.

$$\mathsf{edge}(x,y) \Rightarrow \mathsf{path}(x,y)$$
$$\mathsf{edge}(x,y) \wedge \mathsf{path}(y,z) \Rightarrow \mathsf{path}(x,z)$$

3.1 Substructural Connectives

Since you have seen substructural inference, however, you may observe that some of these principles don't apply to how we used them to modify knowledge states. In particular, for structural inference, we always *added* things to the context as we work upward, whereas in linear inference we want to *replace* them:

$$\frac{\Gamma, \mathsf{wood}, \mathsf{wood}, \mathsf{wood}, \mathsf{wood} \vdash C}{\Gamma, \mathsf{tree} \vdash C}$$

This also makes our original definition of $\wedge$ seem suspect, because we used the *same context* to solve our subgoals. What might make more sense for linear logic? We will use a different symbol $\otimes$ (tensor) to represent this kind of conjunction, which is meant to represent combining linear resources (internalizing the linear logic version of ",").

$$\frac{\Delta_1 \vdash A \quad \Delta_2 \vdash B}{\Delta_1\ \Delta_2 \vdash A \otimes B}$$

That is, to prove that we can get to a goal consisting of both A *and* B, simultaneously, we need to use part of our resources to get A and part of our resources to get B.

Actually, we *could* use the previous definition of $\wedge$ in linear logic (usually we write in &). What would it mean?

We can use it to represent the fact that our same initial state actually makes it possible to derive multiple different goals—recall how one of our forward inference chains got stuck, and we wound up somewhere we didn't want to be. We can characterize this kind of nondeterminism with the & connective.

4 The Possibility Space of Substructural Logics

Now that we have sequent notation, we can finally give formal definitions to the *structural rules*:

$$\frac{\Gamma \vdash C}{\Gamma, B \vdash C}\ \mathsf{weaken}$$

$$\frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ \mathsf{contract}$$

$$\frac{\Gamma, \Gamma' \vdash C}{\Gamma', \Gamma \vdash C}\ \mathsf{exchange}$$

In *structural inference*, contexts permit all of the structural rules (either implicitly or by fiat), and substrucural contexts do not. The inclusion or exclusion of these rules determines whether a valid proof (program) must respect the ordering of assumptions as we use them, and how many times we can use them. Linear logic, by forgoing weakening and contraction, requires every assumption to be used exactly once; affine logic forgoes only contraction (use each assumption zero or one times); strict logic only weakening (use each assumption at least once); and ordered logic forgoes all three. We've currently investigated only two of these—linear and ordered logic—but you now have the conceptual tools to think about the others.

An alternate view is that each of these logics can be thought of as modeling the context with different algebraic structures. Structural logic's contexts behave like sets; linear logical contexts behave like commutative monoids; ordered logical contexts like (noncommutative) monoids. (For those with an interest in category theory, there is also a useful story about interpreting these context structures into different categories and getting their different connectives via the same categorical constructions; you can ask me about that later if you're curious.)

Here's a table summarizing the differences between logics in terms of structural rules, as witnessed by their multiplicative conjunction forms:

	Associativity	Commutativity	Idempotence
SL	$A \wedge (B \wedge C) \equiv (A \wedge B) \wedge C$	$A \wedge B \equiv B \wedge A$	$A \wedge A \equiv A$
LL	$A \otimes (B \otimes C) \equiv (A \otimes B) \otimes C$	$A \otimes B \equiv B \otimes A$	$A \otimes A \not\equiv A$
OL	$A \bullet (B \bullet C) \equiv (A \bullet B) \bullet C$	$A \bullet B \not\equiv B \bullet A$	$A \bullet A \not\equiv A$

(You can read $A \equiv B$ as "bi-entailment", i.e. $A \vdash B$ and $B \vdash A$.)

One might wonder if we could drop structural equivalences even further, eschewing associativity, in which case the context is a magma. This gives us *rigid logic*. Developing its connectives (and computational potential) is left as an exercise to the reader.

It is also fruitful to consider combinations of the structural rules not given here, such as dropping contraction but not weakening or weakening but not contraction. Dropping exchange is only meaningful when you also drop both weakening and contraction; otherwise, you can recover exchange (proving this is an exercise for the reader).

4.1 Combining Logics

So, you may be asking yourself, what is the point of writing down all these different kinds of symbol-pushing? One answer is that they follow different rules, so they model different kinds of problems. The idea is that these logics give us more expressive power by *making finer distinctions* between the kinds of programs we might want to write. But to really make use of these features, we need to be able to *combine* their powers. After all, we don't want a typechecker to force *all* of our functions to use their arguments exactly once!

If want to be able to use all of modes of truth represented in these logics in one setting, then we need to combine them. One way to do this is to start off in the "most restrictive" logic you want to use (say, ordered logic), and "import" propositions from the less-restrictive logics. We can do this with *modalities*: a structural logic proposition A_s can be mapped to OL with $!A_s$ (the "bang" modality), and a linear (mobile, from the OL point of view) proposition A_l can be mapped with ¡A_l (the "gnab" modality). (There is another way to do this, that also generalizes to logics beyond substructural ones, based on a category theoretic concept called *adjoint functors*, which you can read up on [5] if you're curious!)

5 Conclusion

We have explored the idea of substructural logics first through the idea of modeling inference about different kinds of systems (path enumeration in graphs; resource exchange; syntax parsing), then by developing logical connectives and their corresponding proof terms, which give rise to different computational interpretations. Different ways of defining the same logical connectives give us distinct computational interpretations, which in turn give us more precise ways of exposing to programmers the distinct mathematical structures that underlie computational problems.

References

[1] J.-Y. Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[2] J.-Y. Girard. On the unity of logic. *Annals of pure and applied logic*, 59(3):201–217, 1993.

[3] J. Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65(3):154–170, 1958.

[4] P. B. Levy. *Call-by-push-value.* PhD thesis, 2013.

[5] K. Pruiksma, W. Chargin, F. Pfenning, and J. Reed. Adjoint logic. *Unpublished manuscript, April*, 2018.

A Things I didn't talk about: Functional Programming

A.1 Introduction

Practitioners and theorists of functional programming may be familiar with the common typing judgment for typed lambda calculi, often read "expression e has type τ under context Γ":

$$\Gamma \vdash e : \tau$$

The presence of the context Γ, which usually consists of *variable typing assignments* $x_1{:}\tau_1 \ldots x_n{:}\tau_n$, allows us to typecheck expressions that contain variables within the appropriate scope. This also allows us to describe the operation of *functions*, or lambda abstraction, as taking arguments which can then be added to the context as variables to type their bodies:

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.\, e : \tau \to \tau'}$$

This talk concerns the nature of contexts Γ and the computational content of different decisions one can make about its structure. For example, by default, one would expect the function

$$\lambda x{:}A.\, \langle x, x \rangle$$

to be well-typed at $A \to A \times A$, and likewise the function

$$\lambda x{:}A.\, \lambda y{:}B.\, y$$

at $A \to B \to B$. However, each of these type assignments relies on certain facts about how variables added to the context are accessed, called the *structural rules* (because they define the structure of contexts, rather than of specific type assignments).

$$\frac{\Gamma \vdash C}{\Gamma, B \vdash C}\ \textsf{weaken}$$

$$\frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ \textsf{contract}$$

$$\frac{\Gamma, \Gamma' \vdash C}{\Gamma', \Gamma \vdash C}\ \textsf{exchange}$$

The first example relies on *contraction*, which is that we can access variables multiple times in expressions here they are in scope. The second uses *weakening*, which is the idea that we can ignore assumptions (or, alternately, if an expression is well-typed in Γ, it is also well-typed in any *extension* of Γ, such as $\Gamma, x{:}B$). Finally, we also expect that we can use variables in any *order* that they are introduced by function abstraction, which corresponds to the notion of *exchange*, i.e. that contexts (while we may write them or implement them as lists) are actually order-agnostic.

Substructural logics are logics—and by extension, type systems—that eschew one or more of these structural rules.

A.2 Logical Connectives

We can turn our path inference rules into the following *proposition*:

$$\forall x, y, w.\ \mathsf{edge}(x, y) \vee (\mathsf{path}(x, w) \wedge \mathsf{path}(w, y)) \Rightarrow \mathsf{path}(x, y)$$

This proposition uses the *connectives* $\forall$ (universal quantification), $\vee$ (disjunction), $\wedge$ (conjunction), and $\Rightarrow$ (implication), along with *propositional atoms* $\mathsf{edge}(x, y)$ and $\mathsf{path}(x, y)$ (defined by the "domain"). Each of these connectives *internalizes* some idea from the meta-logic of inference rules: universal quantification captures the idea of generalizing rules over term variables (here, the nodes of the graph); disjunctions, the presence of multiple rules that can derive the same fact; conjunction, the idea of requiring multiple rule-premises; and implication, the concept of inference from premises.

We don't have time to get into how all of these connectives are defined, but to provide a taste, consider the following two rules which can be used to define ($\wedge$):

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \mathsf{pair} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \pi_1 \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \pi_2$$

These rules explain both how to *construct* proofs of $A \wedge B$ (the pair rule) and how to *use* them (the π_1 and π_2 rules), all *in terms of the machinery of inference rules* and not by involving any other connectives. This idea of defining connectives in terms of *introductions* and *eliminators* is key to understanding the correspondence between proofs an programs: you may already be able to see from these suggestive rule names how these proofs correspond to typing rules for pairs and projections in a language with product types.

We also need to give a rule for using an assumption in the context:

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash A}\ x$$

This is known as the *hypothesis* rule. With these four rules, we can see how to construct a simple proof ("conjunction is commutative"):

$$\frac{\dfrac{\dfrac{}{x{:}A \wedge B \vdash A \wedge B}\ x}{x{:}A \wedge B \vdash B}\ \pi_2 \qquad \dfrac{\dfrac{}{x{:}A \wedge B \vdash A \wedge B}\ x}{x{:}A \wedge B \vdash A}\ \pi_1}{x{:}A \wedge B \vdash B \wedge A}\ \mathsf{pair}$$

Finally, if you use a kind of similar notation to our "compact" proof from before, the correspondence with programs is even more obvious:

$$x{:}A \wedge B \vdash \mathsf{pair}(\pi_2(x), \pi_1(x)) : B \wedge A$$

A.3 Substructural Connectives

We would like to make these different kinds of inference formal in terms of how we define our logic and its connectives. Let's first look at the "hypothesis" rule:

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash A}\ x$$

If we want to not treat contexts as sets, we should refine this rule to

$$\frac{}{x{:}A \vdash A}\ x$$

That is, we can conclude A if it is the *only* thing in our context.

Since we saw how *and* ($\wedge$) was defined for structural logic, let's see if we can define it more generally for the substructural cases. We'll use a different context metavariable Δ this time:

$$\frac{??? \vdash A \quad ??? \vdash B}{\Delta \vdash A \wedge B}$$

Now we can see a choice: should the context in which we prove A and B be the same as the context in which we conclude their conjunction? Or should the conjunction use some kind of combination of the two?

If we want conjunction to *internalize* the structure of contexts, we should write the rules as follows, using $\otimes$ to represent this kind of conjunction (pronounced "tensor"):

$$\frac{\Delta_1 \vdash A \quad \Delta_2 \vdash B}{\Delta_1\ \Delta_2 \vdash A \otimes B}\ \otimes\mathsf{I} \qquad \frac{\Delta \vdash A \otimes B \quad \Delta_1\ A\ B\ \Delta_2 \vdash C}{\Delta_1\ \Delta\ \Delta_2 \vdash C}\ \otimes E$$

Ordered logic's context-internalizing conjunction, written $A \bullet B$, can be defined in much the same way. (We elide the full definition of the logic, but the main difference is at the structural level, i.e. how we treat contexts, rather than in the definition of specific connectives.)

An interesting question: what do the proof terms look like?

$$\frac{\Delta_1 \vdash M : A \quad \Delta_2 \vdash N : B}{\Delta_1\ \Delta_2 \vdash \{M, N\} : A \otimes B}\ \otimes\mathsf{I} \qquad \frac{\Delta \vdash M : A \otimes B \quad \Delta_1\ x{:}A\ y{:}B\ \Delta_2 \vdash N_{x,y} : C}{\Delta_1\ \Delta\ \Delta_2 \vdash \mathsf{let}\ \{x, y\} = M\ \mathsf{in}\ N : C}\ \otimes E$$

This proof term assignment might remind you of *pattern matching* on pairs in a functional language. This is not mere coincidence!

It turns out we can *also* write a rule for a connective defined more closely to the structural version of "and", and get a different kind of conjunction. We'll write this one &, pronounced "with":

$$\frac{\Delta \vdash A \quad \Delta \vdash B}{\Delta \vdash A \& B}\ \& I \qquad \frac{\Delta \vdash A \& B}{\Delta \vdash A}\ \& E_1 \qquad \frac{\Delta \vdash A \& B}{\Delta \vdash B}\ \& E_2$$

We call this the *additive* conjunction, as opposed to *multiplicative* conjunction ($\otimes$).

What does the & connective represent in terms of substructural logics, e.g. in our specific examples?

If we're using ordered logic to represent grammatical parses, this corresponds to an *ambiguous parse.* That is, a given phrase represented in Δ could parse as A or as B. In linear logic, it represents multiple ways to use the same set of resources, e.g. spending the same ten dollars to purchase different items. In general, $A\&B$ represents the possibility of drawing multiple conclusions from the same set of premises.

The idea of being able to represent conjunction in these two different ways gives us insight about the nature of structural logic's $\wedge$ that we wouldn't have otherwise: in particular, the idea of *pattern matching* on pairs is sound (in structural logic!) with respect to the alternate definition using projections. It may be illustrative to write proof terms witnessing the fact that, if we treat contexts structurally, $A\&B \equiv A \otimes B$. (You can read $A \equiv B$ as "bi-entailment", i.e. $A \vdash B$ and $B \vdash A$.) Substructural logics, by separating these two definitions into distinct logical connectives, paves the way to understanding a concept called *polarity* [2], which turns out to be a deep theory about the nature of program structure, pattern matching, and evaluation order (also known as call-by-push-value [4]).

Apart from conjunction, other substructural connectives one can define include linear implication ($\multimap$), *two* kinds of ordered implication (depending on whether the argument is expected to the left or right) ($\rightarrowtail$, $\twoheadrightarrow$), a substructural version of disjunction ($\oplus$), and *units* for all of the binary connectives ($\mathbb{1}$ for $\otimes$; $\top$ for $\&$; $\mathbb{0}$ for $\oplus$).