Cool Logic Systems

(Substructural Logic and Linear Type Systems)

Hype for Types

February 9, 2026

What We'll Talk About

- What it means for a logic to be "substructural"

What We'll Talk About

- What it means for a logic to be "substructural"
- A case study of a particular substructural logic (linear logic)

What We'll Talk About

- What it means for a logic to be "substructural"
- A case study of a particular substructural logic (linear logic)
- Ok this is cool, but does this work in the real world? (If we have time)

Substructural Logic

What Does it Mean to be Substructural?

The constructive logic we have been working with so far has the following admissible rules, which we call "structural properties" of the logic:

$$\frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ (\textsc{Weak}) \qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ (\textsc{Cntr})$$

$$\frac{\Gamma, A, B \vdash C}{\Gamma, B, A \vdash C}\ (\textsc{Exch})$$

What Does it Mean to be Substructural?

What happens if you remove some of these structural properties?

What Does it Mean to be Substructural?

What happens if you remove some of these structural properties? You would get a new logical system!

What Does it Mean to be Substructural?

What happens if you remove some of these structural properties? You would get a new logical system!

- Affine Logic: no contraction (Use premises at most once)

What Does it Mean to be Substructural?

What happens if you remove some of these structural properties? You would get a new logical system!

- Affine Logic: no contraction (Use premises at most once)
- Relevance Logic: no weakening (Use premises at least once)

What Does it Mean to be Substructural?

What happens if you remove some of these structural properties? You would get a new logical system!

- Affine Logic: no contraction (Use premises at most once)
- Relevance Logic: no weakening (Use premises at least once)
- Linear Logic: no weakening or contraction (Use premises exactly once)

What Does it Mean to be Substructural?

What happens if you remove some of these structural properties? You would get a new logical system!

- Affine Logic: no contraction (Use premises at most once)
- Relevance Logic: no weakening (Use premises at least once)
- Linear Logic: no weakening or contraction (Use premises exactly once)
- Ordered Logic: no weakening, contraction, or exchange (Use premises exactly once and order matters)

What Does it Mean to be Substructural?

What happens if you remove some of these structural properties? You would get a new logical system!

- Affine Logic: no contraction (Use premises at most once)
- Relevance Logic: no weakening (Use premises at least once)
- Linear Logic: no weakening or contraction (Use premises exactly once)
- Ordered Logic: no weakening, contraction, or exchange (Use premises exactly once and order matters)

Question

What are the consequences of not having these structural properties?

What Does it Mean to be Substructural?

What happens if you remove some of these structural properties? You would get a new logical system!

- Affine Logic: no contraction (Use premises at most once)
- Relevance Logic: no weakening (Use premises at least once)
- Linear Logic: no weakening or contraction (Use premises exactly once)
- Ordered Logic: no weakening, contraction, or exchange (Use premises exactly once and order matters)

Question

What are the consequences of not having these structural properties?

Today, we'll be focusing on *linear logic*, and how we can relax it a little bit to get a very useful programming language.

Linear Logic

Is Logic Logical?

The moon is made of green cheese. Therefore, you come to hype for types today.

Question

Is this logical?

Different Interpretation of Implication

Constructive logic interprets $A \Rightarrow B$ as "If you give me A is true, then I give you B is true."
But what it really says is "If you give me as many copy of A as I need, then I give you B is true."

Different Interpretation of Implication

Constructive logic interprets $A \Rightarrow B$ as "If you give me A is true, then I give you B is true."
But what it really says is "If you give me as many copy of A as I need, then I give you B is true."

Idea

The problem of previous example is that to prove the conclusion we only need zero copies of the assumption, hence lacking "relevance".

Different Interpretation of Implication

Constructive logic interprets $A \Rightarrow B$ as "If you give me A is true, then I give you B is true."
But what it really says is "If you give me as many copy of A as I need, then I give you B is true."

Idea

The problem of previous example is that to prove the conclusion we only need zero copies of the assumption, hence lacking "relevance".

Idea

We need a logic that forces relevance.

Is Logic Logical?

You and your friends go to the hip new spot: the constructive logic cafe! You have \$10. On theme, the menu says:

- \$6 $\Rightarrow$ *Coffee*
- \$6 $\Rightarrow$ *Muffin*

[1] You have a lot of friends

Is Logic Logical?

You and your friends go to the hip new spot: the constructive logic cafe! You have \$10. On theme, the menu says:

- \$6 $\Rightarrow$ *Coffee*
- \$6 $\Rightarrow$ *Muffin*

To the owner's dismay, bound by the laws of constructive logic, you walk out with their entire stock of muffins and a coffee.[1]

Question

Is this logical?

[1] You have a lot of friends

Is Logic Logical?

You and your friends go to the hip new spot: the constructive logic cafe! You have \$10. On theme, the menu says:

- \$6 $\Rightarrow$ *Coffee*
- \$6 $\Rightarrow$ *Muffin*

To the owner's dismay, bound by the laws of constructive logic, you walk out with their entire stock of muffins and a coffee.[1]

Question

Is this logical?

Idea

The problem is that "regular" logic allows us to freely duplicate assumptions.

Idea

We need a logic that limits usage.

[1] You have a lot of friends

Malloc is Scary...

Consider the following C code:

```c
int main () {
  char *str;
  str = (char *) malloc(13);
  strcpy(str, "hypefortypes");
  free(str);
  return(0);
}
```

In C, we have to make sure we allocate and deallocate every memory cell exactly once.

Malloc is Scary...

Consider the following C code:

```c
int main () {
  char *str;
  str = (char *) malloc(13);
  strcpy(str, "hypefortypes");
  free(str);
  return(0);
}
```

In C, we have to make sure we allocate and deallocate every memory cell exactly once.

Question

Is there a way to make our *types* guarantee correctness?

The Problem With Constructive Logic

In “normal” constructive logic, we have no concept of *state*.

The Problem With Constructive Logic

In “normal” constructive logic, we have no concept of *state*.

Big Idea

Proofs should no longer be *persistent*, but rather *ephemeral*.

The Problem With Constructive Logic

In "normal" constructive logic, we have no concept of *state*.

Big Idea

Proofs should no longer be *persistent*, but rather *ephemeral*.

Persistence is due to implicit **structural rules**: weakening and contraction.

Weakening

```c
int main() {
  int *x = (int *) malloc(sizeof(int));
  *x = 3;
  return 0;
}
```

Weakening

```c
int main() {
  int *x = (int *) malloc(sizeof(int));
  *x = 3;
  return 0;
}
```

Weakening: we can "drop" assumptions

$$\frac{\Gamma \vdash e : \tau}{\Gamma, x : \tau' \vdash e : \tau} \ (\textsc{Weak})$$

Contraction

```c
void f(int *x) {
  free(x);
}

int main() {
  int *x = (int *) malloc(sizeof(int));
  *x = 3;
  f(x);
  f(x);
  return 0;
}
```

Contraction

```c
void f(int *x) {
  free(x);
}

int main() {
  int *x = (int *) malloc(sizeof(int));
  *x = 3;
  f(x);
  f(x);
  return 0;
}
```

Contraction: we can “duplicate” assumptions

$$\frac{\Gamma, x_1 : \tau, x_2 : \tau \vdash e : \tau'}{\Gamma, x : \tau \vdash [x, x / x_1, x_2]e : \tau'} \ (\textsc{Cntr})$$

Introduction to Linear Logic

In **linear logic**, we have neither weakening nor contraction.

- Requirement that we use each piece of data *exactly* once - no duplication, no dropping
- Comes with an inherent idea of “resources” that are used up
- Allows us to write safe, stateful (imperative!) programs

The Linear Rules

Identity

Constructive Logic

$$\frac{A \in \Gamma}{\Gamma \vdash A} \; (\textsc{Hyp})$$

Identity

Constructive Logic

$$\frac{A \in \Gamma}{\Gamma \vdash A} \ (\textsc{Hyp})$$

Linear Logic

$$\frac{}{A \vdash A} \ (\textsc{Hyp})$$

Identity

Constructive Logic

$$\frac{A \in \Gamma}{\Gamma \vdash A}\ (\textsc{Hyp})$$

Linear Logic

$$\frac{}{A \vdash A}\ (\textsc{Hyp})$$

Intuition

"Given A and nothing else, we can use up A"

Conjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \wedge A_2}\ (\wedge\mathrm{I})$$

$$\frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_1}\ (\wedge\mathrm{E1}) \qquad\qquad \frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_2}\ (\wedge\mathrm{E2})$$

Conjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \wedge A_2}\ (\wedge\mathrm{I})$$

$$\frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_1}\ (\wedge\mathrm{E1}) \qquad\qquad \frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_2}\ (\wedge\mathrm{E2})$$

Linear Logic

Conjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \wedge A_2} \ (\wedge\mathrm{I})$$

$$\frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_1} \ (\wedge\mathrm{E1}) \qquad\qquad \frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_2} \ (\wedge\mathrm{E2})$$

Linear Logic

$$\frac{\Delta_1 \vdash A_1 \qquad \Delta_2 \vdash A_2}{\Delta_1, \Delta_2 \vdash A_1 \otimes A_2} \ (\otimes\mathrm{I})$$

Conjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \land A_2} \ (\land\mathrm{I})$$

$$\frac{\Gamma \vdash A_1 \land A_2}{\Gamma \vdash A_1} \ (\land\mathrm{E1}) \qquad\qquad \frac{\Gamma \vdash A_1 \land A_2}{\Gamma \vdash A_2} \ (\land\mathrm{E2})$$

Linear Logic

$$\frac{\Delta_1 \vdash A_1 \qquad \Delta_2 \vdash A_2}{\Delta_1, \Delta_2 \vdash A_1 \otimes A_2} \ (\otimes\mathrm{I})$$

$$\frac{\Delta \vdash A_1 \otimes A_2 \qquad \Delta', A_1, A_2 \vdash C}{\Delta, \Delta' \vdash C} \ (\otimes\mathrm{E})$$

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_1) \qquad \frac{\Gamma \vdash A_2}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_2)$$

$$\frac{\Gamma \vdash A_1 \vee A_2 \qquad \Gamma, A_1 \vdash B \qquad \Gamma, A_2 \vdash B}{\Gamma \vdash B}\ (\vee E)$$

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_1) \qquad \frac{\Gamma \vdash A_2}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_2)$$

$$\frac{\Gamma \vdash A_1 \vee A_2 \qquad \Gamma, A_1 \vdash B \qquad \Gamma, A_2 \vdash B}{\Gamma \vdash B}\ (\vee E)$$

Linear Logic

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_1) \qquad \frac{\Gamma \vdash A_2}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_2)$$

$$\frac{\Gamma \vdash A_1 \vee A_2 \qquad \Gamma, A_1 \vdash B \qquad \Gamma, A_2 \vdash B}{\Gamma \vdash B}\ (\vee E)$$

Linear Logic

$$\frac{\Delta \vdash A_1}{\Delta \vdash A_1 \oplus A_2}\ (\oplus \mathrm{I1})$$

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_1) \qquad \frac{\Gamma \vdash A_2}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_2)$$

$$\frac{\Gamma \vdash A_1 \vee A_2 \qquad \Gamma, A_1 \vdash B \qquad \Gamma, A_2 \vdash B}{\Gamma \vdash B}\ (\vee E)$$

Linear Logic

$$\frac{\Delta \vdash A_1}{\Delta \vdash A_1 \oplus A_2}\ (\oplus\mathrm{I1}) \qquad \frac{\Delta \vdash A_2}{\Delta \vdash A_1 \oplus A_2}\ (\oplus\mathrm{I2})$$

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_1) \qquad\qquad \frac{\Gamma \vdash A_2}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_2)$$

$$\frac{\Gamma \vdash A_1 \vee A_2 \qquad \Gamma, A_1 \vdash B \qquad \Gamma, A_2 \vdash B}{\Gamma \vdash B}\ (\vee E)$$

Linear Logic

$$\frac{\Delta \vdash A_1}{\Delta \vdash A_1 \oplus A_2}\ (\oplus \mathrm{I1}) \qquad\qquad \frac{\Delta \vdash A_2}{\Delta \vdash A_1 \oplus A_2}\ (\oplus \mathrm{I2})$$

$$\frac{\Delta \vdash A_1 \oplus A_2 \qquad \Delta', A_1 \vdash B \qquad \Delta', A_2 \vdash B}{\Delta, \Delta' \vdash B}\ (\oplus \mathrm{E})$$

Implication

Constructive Logic

$$\frac{\Gamma, A_1 \vdash A_2}{\Gamma \vdash A_1 \supset A_2} \; (\supset I) \qquad\qquad \frac{\Gamma \vdash A_1 \supset A_2 \quad \Gamma \vdash A_1}{\Gamma \vdash A_2} \; (\supset E)$$

Implication

Constructive Logic

$$\frac{\Gamma, A_1 \vdash A_2}{\Gamma \vdash A_1 \supset A_2}\ (\supset I) \qquad\qquad \frac{\Gamma \vdash A_1 \supset A_2 \quad \Gamma \vdash A_1}{\Gamma \vdash A_2}\ (\supset E)$$

Linear Logic

Implication

Constructive Logic

$$\frac{\Gamma, A_1 \vdash A_2}{\Gamma \vdash A_1 \supset A_2} \ (\supset I) \qquad \frac{\Gamma \vdash A_1 \supset A_2 \quad \Gamma \vdash A_1}{\Gamma \vdash A_2} \ (\supset E)$$

Linear Logic

$$\frac{\Delta, A_1 \vdash A_2}{\Delta \vdash A_1 \multimap A_2} \ (\multimap I)$$

Implication

Constructive Logic

$$\frac{\Gamma, A_1 \vdash A_2}{\Gamma \vdash A_1 \supset A_2} \ (\supset I) \qquad \frac{\Gamma \vdash A_1 \supset A_2 \quad \Gamma \vdash A_1}{\Gamma \vdash A_2} \ (\supset E)$$

Linear Logic

$$\frac{\Delta, A_1 \vdash A_2}{\Delta \vdash A_1 \multimap A_2} \ (\multimap I) \qquad \frac{\Delta \vdash A_1 \multimap A_2 \quad \Delta' \vdash A_1}{\Delta, \Delta' \vdash A_2} \ (\multimap E)$$

Model Real Worlds Using Linear Logic

Recall the Constructive Logic Cafe

- $\$6 \Rightarrow \text{Coffee}$
- $\$6 \Rightarrow \text{Muffin}$

We have \$10 to spend and we are both hungry and thirsty.

Model Real Worlds Using Linear Logic

Recall the Constructive Logic Cafe

- \$6 $\Rightarrow$ Coffee
- \$6 $\Rightarrow$ Muffin

We have \$10 to spend and we are both hungry and thirsty.

Stuctural Logic Causes Inflation!

Well $(A \Rightarrow B) \wedge (A \Rightarrow C) \Rightarrow (A \Rightarrow B \wedge C)$ so we can buy both! This doesn't seem right...

Time for a Rebrand!

Rebranding to the Linear Logic Cafe[a]

[a] LLC LLC

- $\$6 \multimap \text{Coffee}$
- $\$6 \multimap \text{Muffin}$

Time for a Rebrand!

Rebranding to the Linear Logic Cafe[a]

[a] LLC LLC

- \$6 $\multimap$ Coffee
- \$6 $\multimap$ Muffin

Linear Logic Does Not

$(A \multimap B) \otimes (A \multimap C) \not\multimap (A \multimap B \otimes C)$ so we cannot buy both! Capitalism is saved![a]

[a] What have we done...

The Actual Real World

You Can Take My Structural Rules From My Cold, Dead Hands!

Imagine a substructural programming language... what would that look like?

You Can Take My Structural Rules From My Cold, Dead Hands!

Imagine a substructural programming language... what would that look like?

Complaints

- I like using variables more than once!

You Can Take My Structural Rules From My Cold, Dead Hands!

Imagine a substructural programming language... what would that look like?

Complaints

- I like using variables more than once!
- I want to easily forget about some values (e.g. after an effectful computation)

You Can Take My Structural Rules From My Cold, Dead Hands!

Imagine a substructural programming language... what would that look like?

Complaints

- I like using variables more than once!
- I want to easily forget about some values (e.g. after an effectful computation)
- You're telling me the order I declare variables should matter!??

You Can Take My Structural Rules From My Cold, Dead Hands!

Imagine a substructural programming language... what would that look like?

Complaints

- I like using variables more than once!
- I want to easily forget about some values (e.g. after an effectful computation)
- You're telling me the order I declare variables should matter!??

These are fair criticisms! So let's compromise!

Affine is key!

Your criticisms are valid! I'll give you back normal identity rules, and therefore weakening:

Weakening is nice to have

$$\frac{A \in \Gamma}{\Gamma \vdash A}\ (\textsc{Id}) \qquad \frac{\Gamma \vdash e : \tau}{\Gamma, x : \tau' \vdash e : \tau}\ (\textsc{Weak})$$

Affine is key!

Your criticisms are valid! I'll give you back normal identity rules, and therefore weakening:

Weakening is nice to have

$$\frac{A \in \Gamma}{\Gamma \vdash A}\ (\textsc{Id}) \qquad \frac{\Gamma \vdash e : \tau}{\Gamma, x : \tau' \vdash e : \tau}\ (\textsc{Weak})$$

And exchange is...

Affine is key!

Your criticisms are valid! I'll give you back normal identity rules, and therefore weakening:

Weakening is nice to have

$$\frac{A \in \Gamma}{\Gamma \vdash A}\ (\textsc{Id}) \qquad \frac{\Gamma \vdash e : \tau}{\Gamma, x : \tau' \vdash e : \tau}\ (\textsc{Weak})$$

And exchange is... yeah you can have that too

Affine is key!

Your criticisms are valid! I'll give you back normal identity rules, and therefore weakening:

Weakening is nice to have

$$\frac{A \in \Gamma}{\Gamma \vdash A}\ (\textsc{Id}) \qquad \frac{\Gamma \vdash e : \tau}{\Gamma, x : \tau' \vdash e : \tau}\ (\textsc{Weak})$$

And exchange is... yeah you can have that too
But, can we negotiate on contraction?

Recall Contraction

Contraction: The Idea

$$\frac{\Gamma, x_1 : \tau, x_2 : \tau \vdash e : \tau'}{\Gamma, x : \tau \vdash [x, x/x_1, x_2]e : \tau'} \ (\textsc{Cntr})$$

Recall Contraction

Contraction: The Idea

$$\frac{\Gamma, x_1 : \tau, x_2 : \tau \vdash e : \tau'}{\Gamma, x : \tau \vdash [x, x/x_1, x_2]e : \tau'} \ (\textsc{Cntr})$$

We can use our variables more than once.

Recall Contraction

Contraction: The Idea

$$\frac{\Gamma, x_1 : \tau, x_2 : \tau \vdash e : \tau'}{\Gamma, x : \tau \vdash [x, x/x_1, x_2]e : \tau'} \ (\textsc{Cntr})$$

We can use our variables more than once.

Why not contraction?

Benefits!

- We can never double free!

Recall Contraction

Contraction: The Idea

$$\frac{\Gamma, x_1 : \tau, x_2 : \tau \vdash e : \tau'}{\Gamma, x : \tau \vdash [x, x/x_1, x_2]e : \tau'} \ (\textsc{Cntr})$$

We can use our variables more than once.

Why not contraction?

Benefits!

- We can never double free!
- We cannot write race conditions!

Recall Contraction

Contraction: The Idea

$$\frac{\Gamma, x_1 : \tau, x_2 : \tau \vdash e : \tau'}{\Gamma, x : \tau \vdash [x, x/x_1, x_2]e : \tau'} \ (\textsc{Cntr})$$

We can use our variables more than once.

Why not contraction?

Benefits!

- We can never double free!
- We cannot write race conditions!
- We get automatic memory management without the cost of GC!

Applications

Blocks World: Planning as Proof Search

- A robot arm moves blocks on a table: pick up / put down one block at a time.
- Model a **world-state** as a collection of facts (resources).
- A **plan** is a proof that transforms an initial state into a goal.

Typical facts (resources)

$\mathbf{on}(x, y)$, $\mathbf{tb}(x)$, $\mathbf{holds}(x)$, $\mathbf{empty}$, $\mathbf{clear}(x)$

Why Ordinary Implication Breaks "State"

A tempting (but wrong) encoding of an action is:

$$\forall x.\forall y.\ (\mathit{empty} \wedge \mathit{clear}(x) \wedge \mathit{on}(x, y)) \supset (\mathit{holds}(x) \wedge \mathit{clear}(y))$$

Why Ordinary Implication Breaks "State"

A tempting (but wrong) encoding of an action is:

$$\forall x.\forall y.\ (\mathit{empty} \wedge \mathit{clear}(x) \wedge \mathit{on}(x, y)) \supset (\mathit{holds}(x) \wedge \mathit{clear}(y))$$

Problem

In ordinary logic, assumptions *persist*. So after "picking up", you can still keep *empty*, and derive impossible states like $\mathit{empty} \wedge \mathit{holds}(b)$.

Why Ordinary Implication Breaks "State"

A tempting (but wrong) encoding of an action is:

$$\forall x. \forall y.\ (\mathit{empty} \wedge \mathit{clear}(x) \wedge \mathit{on}(x, y)) \supset (\mathit{holds}(x) \wedge \mathit{clear}(y))$$

Problem

In ordinary logic, assumptions *persist*. So after "picking up", you can still keep *empty*, and derive impossible states like $\mathit{empty} \wedge \mathit{holds}(b)$.

- Temporal logic can avoid direct contradiction, but then you must also encode "everything else stays the same" (frame conditions).
- Linear logic gets "state change" by making facts *consumable*.

Linear Actions: Resources In, Resources Out

Legal moves become **linear implications** (consume LHS, produce RHS):

$$\textbf{geton} : \forall x.\forall y.\ (\mathit{empty} \otimes \mathit{clear}(x) \otimes \mathit{on}(x,y)) \multimap (\mathit{holds}(x) \otimes \mathit{clear}(y))$$

$$\textbf{puton} : \forall x.\forall y.\ (\mathit{holds}(x) \otimes \mathit{clear}(y)) \multimap (\mathit{empty} \otimes \mathit{on}(x,y) \otimes \mathit{clear}(x))$$

Linear Actions: Resources In, Resources Out

Legal moves become **linear implications** (consume LHS, produce RHS):

$$\textbf{geton} : \forall x.\forall y.\ (\mathit{empty} \otimes \mathit{clear}(x) \otimes \mathit{on}(x,y)) \multimap (\mathit{holds}(x) \otimes \mathit{clear}(y))$$

$$\textbf{puton} : \forall x.\forall y.\ (\mathit{holds}(x) \otimes \mathit{clear}(y)) \multimap (\mathit{empty} \otimes \mathit{on}(x,y) \otimes \mathit{clear}(x))$$

Type-theory punchline

Linear hypotheses behave like **capabilities** (use exactly once). That's the same core idea behind linear/affine type systems for memory, file handles, locks, etc.

Rust: Substructural Types With Training Wheels

- Owned heap values (like `String`) behave **affine**:
  - no implicit duplication (no contraction)
  - dropping is allowed (weakening)

Rust: Substructural Types With Training Wheels

- Owned heap values (like `String`) behave **affine**:
  - no implicit duplication (no contraction)
  - dropping is allowed (weakening)
- **Copy** types (like `i32`) behave "unrestricted" (you *can* duplicate).

Rust: Substructural Types With Training Wheels

- Owned heap values (like `String`) behave **affine**:
  - no implicit duplication (no contraction)
  - dropping is allowed (weakening)
- **Copy** types (like `i32`) behave "unrestricted" (you *can* duplicate).
- Borrowing (`&T` vs `&mut T`) is how Rust enforces "no aliasing + mutation" $\Rightarrow$ no data races.

Move = "Used Exactly Once" (No Contraction)

```rust
fn i_eat_strings(_: String) {}

fn main() {
  let x = "hello".to_string();
  let y = x;              // MOVE ownership from x -> y

  // println!("{x}");
  // not allowed: x was "used up" when moved into y
}
```

Move = "Used Exactly Once" (No Contraction)

```rust
fn i_eat_strings(_: String) {}

fn main() {
  let x = "hello".to_string();
  let y = x;            // MOVE ownership from x -> y

  // println!("{x}");
  // not allowed: x was "used up" when moved into y
}
```

Logic connection

`String` acts like a resource: once consumed/moved, you don't get it back for free.

Clone = Explicit Duplication (Controlled Contraction)

```rust
fn i_eat_strings(_: String) {}

fn main() {
  let y = "hello".to_string();

  i_eat_strings(y.clone()); // explicit duplication
  i_eat_strings(y);         // consume the original

  // i_eat_strings(y);
  // not allowed: y already consumed
}
```

Clone = Explicit Duplication (Controlled Contraction)

```rust
fn i_eat_strings(_: String) {}

fn main() {
  let y = "hello".to_string();

  i_eat_strings(y.clone()); // explicit duplication
  i_eat_strings(y);         // consume the original

  // i_eat_strings(y);
  // not allowed: y already consumed
}
```

Why this is nice

Duplication exists, but it's *visible* and usually costs time/memory.

Copy Types = "Unrestricted" Values

```rust
fn i_dont_consume_ints(x: i32) { println!("{x}") }

fn main() {
  let z = 5;                 // Copy type
  i_dont_consume_ints(z);
  i_dont_consume_ints(z);
  // allowed: z implicitly copied
}
```

Copy Types = “Unrestricted” Values

```rust
fn i_dont_consume_ints(x: i32) { println!("{x}") }

fn main() {
  let z = 5;                 // Copy type
  i_dont_consume_ints(z);
  i_dont_consume_ints(z);
  // allowed: z implicitly copied
}
```

Interpretation

Rust splits the world into:

- **linear/affine-ish** (owned heap stuff)
- **unrestricted** (cheap Copy stuff)

Borrowing: Read Sharing Is Fine

```rust
fn i_can_print_references(s: &str) { println!("{s}"); }

fn main() {
  let s = "world".to_string();
  i_can_print_references(&s);
  i_can_print_references(&s);
  // allowed: shared (&) borrows
}
```

Borrowing: Read Sharing Is Fine

```
fn i_can_print_references(s: &str) { println!("{s}"); }

fn main() {
  let s = "world".to_string();
  i_can_print_references(&s);
  i_can_print_references(&s);
  // allowed: shared (&) borrows
}
```

Intuition

Shared borrows are “read-only capabilities” that can be duplicated.

Borrowing: Mutation Requires Uniqueness

```rust
fn i_modify_numbers(n: &mut i32, _: &mut i32) { *n = 5;

fn main() {
  let mut n = 10;
  let mut m = 10;

  // let x = &mut n;
  // let y = &mut n;
  // not allowed: two mutable borrows
  i_modify_numbers(&mut n, &mut m);

  let mut foo = 5;
  let imm_foo = &foo;
  let mut_foo = &mut foo;
  // not allowed: & and &mut at same time
  // println!("{imm_foo}");
}
```

Borrowing: Mutation Requires Uniqueness

```
fn i_modify_numbers(n: &mut i32, _: &mut i32) { *n = 5; }

fn main() {
  let mut n = 10;
  let mut m = 10;

  // let x = &mut n;
  // let y = &mut n;
  // not allowed: two mutable borrows
  i_modify_numbers(&mut n, &mut m);

  let mut foo = 5;
  let imm_foo = &foo;
  let mut_foo = &mut foo;
  // not allowed: & and &mut at same time
  // println!("{imm_foo}");
}
```

No Dangling References (Lifetimes Prevent It)

Big takeaway

"At most one writer, or many readers" $\Rightarrow$ no data races by construction.

No Dangling References (Lifetimes Prevent It)

```rust
fn i_dangle () -> &'static str {
  let s = String::new();
  &s

  // not allowed: returning a reference
  // to data that will be dropped
}
```

No Dangling References (Lifetimes Prevent It)

```
fn i_dangle () -> &'static str {
  let s = String::new();
  &s

  // not allowed: returning a reference
  // to data that will be dropped
}
```

Why this matters

This is the “no use-after-free” version of linear resource discipline.

Rust as "Affine + Borrowing"

- Owned values: **affine** (no implicit duplication; dropping allowed)
- `clone()`: explicit duplication when you really want contraction
- Borrow checker: enforces safe sharing + safe mutation patterns

Rust as “Affine + Borrowing”

- Owned values: **affine** (no implicit duplication; dropping allowed)
- `clone()`: explicit duplication when you really want contraction
- Borrow checker: enforces safe sharing + safe mutation patterns

Bridge back to today

Substructural rules aren't just philosophy: they're *exactly* what makes ownership + memory safety work.

Ordered Logic: When Order Matters

Start from linear logic (no weakening/duplication) and remove **exchange**.

Ordered Logic: When Order Matters

Start from linear logic (no weakening/duplication) and remove **exchange**.

Intuition

Your context is a *sequence* (a word), not a multiset. You must consume resources *in order*.

Ordered Logic: When Order Matters

Start from linear logic (no weakening/duplication) and remove **exchange**.

Intuition

Your context is a *sequence* (a word), not a multiset. You must consume resources *in order*.

- Great for **parsing** and **grammar** (Lambek calculus)
- Great for **state machines** and **protocol-like sequencing**
- Great for **computation as rewriting** (transducers, Turing machines)

Example: Finite-State Transducer = Ordered Inference

One example FST: compress runs of b into a single b.

Example: Finite-State Transducer = Ordered Inference

One example FST: compress runs of b into a single b.
Encode the input word bbaba (reversed, with an endmarker) and initial state:

$$\$ \; a \; b \; a \; b \; b \; q_0$$

Example: Finite-State Transducer = Ordered Inference

One example FST: compress runs of b into a single b.
Encode the input word bbaba (reversed, with an endmarker) and initial state:

$$\$ \ a \ b \ a \ b \ b \ q_0$$

Each transition becomes a local rewrite rule on a *consecutive* block:

$$a \ q_0 \to q_0 \ a \qquad b \ q_0 \to q_1 \ b \qquad b \ q_1 \to q_1 \ \epsilon \qquad \$ \ q_0 \to q_f \ \$ \qquad q_f \to \cdot$$

Example: Finite-State Transducer = Ordered Inference

One example FST: compress runs of b into a single b.
Encode the input word bbaba (reversed, with an endmarker) and initial state:

$$\$ \; a \; b \; a \; b \; b \; q_0$$

Each transition becomes a local rewrite rule on a *consecutive* block:

$$a \; q_0 \to q_0 \; a \qquad b \; q_0 \to q_1 \; b \qquad b \; q_1 \to q_1 \; \epsilon \qquad \$ \; q_0 \to q_f \; \$ \qquad q_f \to \cdot$$

"Running the machine" $=$ keep applying any rule anywhere it matches.

Example: Turing Machine Configurations as Sequences

Represent a tape as a finite ordered context with endmarkers:

$$\$ \ \cdots \ a_{-1} \ q \ a_0 \ a_1 \ \cdots \ \$$$

Here q is the current state and the head "looks right" at a_0.

Example: Turing Machine Configurations as Sequences

Represent a tape as a finite ordered context with endmarkers:

$$\$ \ \cdots \ a_{-1} \ q \ a_0 \ a_1 \ \cdots \ \$$$

Here q is the current state and the head "looks right" at a_0.

One-step transitions = local rewrites

Move right (write a_0', go to q'):

$$q \ a_0 \to a_0' \ q' \qquad (\mathbf{MR})$$

Move left (needs the left neighbor b):

$$b \ q \ a_0 \to q' \ b \ a_0' \qquad (\mathbf{ML})$$

Example: Turing Machine Configurations as Sequences

Represent a tape as a finite ordered context with endmarkers:

$$\$ \ \cdots \ a_{-1} \ q \ a_0 \ a_1 \ \cdots \ \$$$

Here q is the current state and the head "looks right" at a_0.

One-step transitions = local rewrites

Move right (write a_0', go to q'):

$$q \ a_0 \to a_0' \ q' \qquad \textbf{(MR)}$$

Move left (needs the left neighbor b):

$$b \ q \ a_0 \to q' \ b \ a_0' \qquad \textbf{(ML)}$$

Halting can be modeled by erasing a final state: $q_f \to \cdot$.

Ordered $\Rightarrow$ Linear $\Rightarrow$ Intuitionistic (Big Picture)

- **Ordered**: no Weak/Cntr/Exch $\Rightarrow$ contexts are sequences.
- **Linear**: add Exchange $\Rightarrow$ contexts are multisets.
  - the two directional implications collapse: $A \backslash B$ and $B/A \rightsquigarrow A \multimap B$
  - order-sensitive products collapse to $\otimes$
- **Intuitionistic**: add Weak + Cntr $\Rightarrow$ "ordinary" assumptions (copy/drop).

Ordered ⇒ Linear ⇒ Intuitionistic (Big Picture)

- **Ordered**: no Weak/Cntr/Exch ⇒ contexts are sequences.
- **Linear**: add Exchange ⇒ contexts are multisets.
  - the two directional implications collapse: $A \backslash B$ and $B/A \rightsquigarrow A \multimap B$
  - order-sensitive products collapse to $\otimes$
- **Intuitionistic**: add Weak + Cntr ⇒ "ordinary" assumptions (copy/drop).

Survey-class moral

Substructural logics are "type systems for assumptions": change the structural rules ⇒ change what programs/proofs can do.

Let's Try It!