

Lecture 25 – Substructural Types

AAA551: Programming Language Theory

Jihyeok Park



2026 Spring

- Flow-Sensitive Types (**occurrence typing**)
- **Propositions** relate variables to types; a **type-result** carries then-/else-propositions and an **object**
- **Latent** propositions on function types let predicates **transmit** facts about their arguments
- T-IF **assumes** the test's proposition per branch; types **narrow**
- The basis of **Typed Racket**; TypeScript, Kotlin, and Flow narrow the **same** way

1. Why Substructural Types?
2. The Structural Rules
A Family of Disciplines
3. A Linear Lambda Calculus
Tagging Types: `lin` vs. `un`
No Contraction: Splitting the Context
No Weakening: The Variable Rule
Abstraction and Containment
Worked Example
4. Rust: Substructural Types in the Wild

1. Why Substructural Types?
2. The Structural Rules
A Family of Disciplines
3. A Linear Lambda Calculus
 - Tagging Types: lin vs. un
 - No Contraction: Splitting the Context
 - No Weakening: The Variable Rule
 - Abstraction and Containment
 - Worked Example
4. Rust: Substructural Types in the Wild

A Variable Is Usually a “Fact”

In type system so far, once a variable is in scope you may use it **as often as you want** – zero times, once, or more. The checker **never counts**.

```
let x = 5;
let a = x + x;      // use x twice  -- fine
let b = x * x;      // use x again  -- fine
                   // never use x  -- also fine
```

This is exactly right for a **fact**. “ x is 5” is a piece of knowledge: once you know it, you can **repeat it** or **ignore it** freely.

But not every value behaves like a fact.

Some Values Are “Resources”

A **resource** behaves more like a **coin** than a fact:

- you can spend it **once** – you cannot spend the **same** coin twice
- once spent, it is **gone** – the old “handle” to it is now worthless.

Real programs are full of resources:

resource	must be used . . .
a file handle	closed once – not twice, not never
a heap allocation	freed once – double-free corrupts memory
a lock	released once
a network socket	closed once

Using a resource **twice** (double-free, use-after-close) or **zero** times (a leak) is a **bug** – yet an ordinary type checker says nothing.

In C, freeing a pointer twice is **well-typed** – the bug appears only at runtime:

```
char* p = malloc(8);
free(p);
free(p);      // double free -- but C reports NO error
```

Rust treats an owned value as a **resource**: assigning **moves** it and the old name becomes **unusable**, so the double-free **cannot even be written**:

```
let s = String::from("hi");
let t = s;           // ownership MOVES s -> t
println!("{}", s);  // error[E0382]: borrow of moved value `s`
```

Counting **how often** a value is used is a **substructural type system**.

1. Why Substructural Types?
2. The Structural Rules
A Family of Disciplines
3. A Linear Lambda Calculus
 - Tagging Types: lin vs. un
 - No Contraction: Splitting the Context
 - No Weakening: The Variable Rule
 - Abstraction and Containment
 - Worked Example
4. Rust: Substructural Types in the Wild

The following three **structural rules** make a variable a **fact**:

- **Exchange** – the **order** of variables does not matter.

$$\frac{\Gamma_1, x:\tau_1, y:\tau_2, \Gamma_2 \vdash e : \tau}{\Gamma_1, y:\tau_2, x:\tau_1, \Gamma_2 \vdash e : \tau} \text{ EXCHANGE}$$

- **Weakening** – a variable may go **unused**.

$$\frac{\Gamma \vdash e : \tau}{\Gamma, x:\tau' \vdash e : \tau} \text{ WEAKENING}$$

- **Contraction** – a variable may be used **many times**.

$$\frac{\Gamma, x:\tau', y:\tau' \vdash e : \tau}{\Gamma, x:\tau' \vdash e[y \mapsto x] : \tau} \text{ CONTRACTION}$$

Removing each structural rule turns freedom into a **discipline**:

drop	lost freedom	new guarantee
Exchange	reorder bindings	used in a fixed order
Weakening	ignore a variable	used at least once (no leaks)
Contraction	duplicate a variable	used at most once (no reuse)

Which rules you keep determines **how many times** a value must be used:

system	Exch.	Weak.	Contr.	use ...
unrestricted (ordinary)	✓	✓	✓	any number of times
affine	✓	✓	✗	at most once
relevant	✓	✗	✓	at least once
linear	✓	✗	✗	exactly once
ordered	✗	✗	✗	exactly once, in order

Rust's ownership system is affine (at most once) to get memory safety.

1. Why Substructural Types?
2. The Structural Rules
A Family of Disciplines
3. A Linear Lambda Calculus
 - Tagging Types: `lin` vs. `un`
 - No Contraction: Splitting the Context
 - No Weakening: The Variable Rule
 - Abstraction and Containment
 - Worked Example
4. Rust: Substructural Types in the Wild

Rather than two separate languages, we **tag** each type with a **qualifier** q saying whether its values are facts or resources:

- $\text{un } P$ – **unrestricted**: a **fact**, copy and drop freely
- $\text{lin } P$ – **linear**: a **resource**, use **exactly once**

Qualifiers	q	::=	$\text{lin} \mid \text{un}$
Pretypes	P	::=	$\text{bool} \mid \tau \rightarrow \tau$
Types	τ	::=	qP (qualified pretype)
Terms	e	::=	$x \mid q \text{true} \mid q \text{false} \mid q \lambda x:\tau. e \mid e e$ $\mid \text{if } e \text{ then } e \text{ else } e$
Contexts	Γ	::=	$\cdot \mid \Gamma, x:\tau$

We write the linear function type $\text{lin}(\tau_1 \rightarrow \tau_2)$ also as $\tau_1 \multimap \tau_2$ (the **lollipop** of linear logic): a **one-shot** function – the function value itself must be used exactly once.

When an expression has **two sub-terms** we must **divide** the context between them, so a linear variable lands on **one side only**:

For example, a function application $e_1 e_2$ has two sub-terms – the function e_1 and its argument e_2 ; the context is split into Γ_1 and Γ_2 for them:

$$\frac{\Gamma_1 \vdash e_1 : q(\tau_1 \rightarrow \tau_2) \quad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1 \circ \Gamma_2 \vdash e_1 e_2 : \tau_2} \text{T-APP}$$

The split $\Gamma = \Gamma_1 \circ \Gamma_2$ sends each **linear** binding to **one** side, but may **share** each **unrestricted** one:

$$\frac{}{\cdot = \cdot \circ \cdot} \quad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x:\text{un } P = (\Gamma_1, x:\text{un } P) \circ (\Gamma_2, x:\text{un } P)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x:\text{lin } P = (\Gamma_1, x:\text{lin } P) \circ \Gamma_2} \quad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x:\text{lin } P = \Gamma_1 \circ (\Gamma_2, x:\text{lin } P)}$$

To enforce no weakening, all **leaf** rules must demand that the **entire** context be unrestricted:

$$\frac{\text{un}(\Gamma)}{\Gamma \vdash q \text{ true} : q \text{ bool}} \text{T-TRUE} \quad \frac{\text{un}(\Gamma)}{\Gamma \vdash q \text{ false} : q \text{ bool}} \text{T-FALSE}$$
$$\frac{\text{un}(\Gamma)}{\Gamma, x:\tau \vdash x : \tau} \text{T-VAR}$$

where $\text{un}(\Gamma)$ means: **every** binding in Γ is unrestricted (un).

Now the two restrictions combine perfectly:

- **splitting** (no contraction) \Rightarrow a linear variable is used **at most once**;
- **this rule** (no weakening) \Rightarrow it is used **at least once**.

Together: every linear variable is used **exactly once**.

A function types its body with the parameter added, and stamps the arrow with a qualifier. The side condition $q(\Gamma)$ enforces one **key invariant**:

$$\frac{q(\Gamma) \quad \Gamma, x:\tau_1 \vdash e : \tau_2}{\Gamma \vdash q \lambda x:\tau_1. e : q(\tau_1 \rightarrow \tau_2)} \text{T-ABS}$$

where $q(\Gamma)$ means: **if the closure is un, then un**(Γ) (a `lin` closure constrains nothing).

Containment invariant

An **unrestricted** value **cannot** contain a **linear** value.

Why? An `un` closure can be **copied**, and copying it copies everything it captured. If it had captured a linear `File`, both copies could `close` it – a **double-close**. So a copyable closure must capture **only** facts.

Worked Example: A Full Derivation

The primitives (read, close are unrestricted, take a **linear** handle):

$$\text{read} : \text{un}(\text{lin } F \rightarrow \text{lin } F) \quad \text{close} : \text{un}(\text{lin } F \rightarrow \text{un } U)$$

where $\text{lin } F$ is a **linear file** type and $\text{un } U$ is an **unrestricted unit** type.

$$\frac{\frac{\text{un}(\cdot)}{\cdot \vdash \text{close} : \text{un}(\text{lin } F \rightarrow \text{un } U)} \quad \frac{\frac{\text{un}(\cdot)}{\cdot \vdash \text{read} : \text{un}(\text{lin } F \rightarrow \text{lin } F)} \quad \frac{\text{un}(\cdot)}{f : \text{lin } F \vdash f : \text{lin } F}}{f : \text{lin } F \vdash \text{read } f : \text{lin } F}}{f : \text{lin } F \vdash \text{close}(\text{read } f) : \text{un } U}$$

- each T-APP **splits** the context; the linear f is sent to **one** branch only (here, always to the right);
- f reaches **exactly one** leaf \Rightarrow used **exactly once**; that leaf's leftover is empty, so $\text{un}(\cdot)$ holds.

1. Why Substructural Types?
2. The Structural Rules
A Family of Disciplines
3. A Linear Lambda Calculus
Tagging Types: lin vs. un
No Contraction: Splitting the Context
No Weakening: The Variable Rule
Abstraction and Containment
Worked Example
4. Rust: Substructural Types in the Wild

Rust's core rule: every value has **exactly one owner**. Binding or passing it **moves** ownership and **invalidates** the old name – use **at most once**.

```
let s = String::from("hi");
let t = s;           // ownership MOVES s -> t
println!("{s}");    // error[E0382]: borrow of moved value `s`
```

Why affine and not linear? Because Rust **keeps weakening**: a value may simply go out of scope. The compiler then inserts its destructor (Drop) for you.

reuse \Rightarrow **rejected**

discard \Rightarrow **allowed (auto-Drop)**

Rust is **affine**: no Contraction (no reuse), but Weakening (drop is fine).

Some values are cheap and harmless to duplicate – `i32`, `bool`, `char`.
These implement the `Copy` trait, so assignment **copies** instead of moving:

```
let x = 5;
let y = x;           // i32 is Copy: x is duplicated, not moved
println!("{x}");    // OK -- x is still valid
```

So the qualifier from our calculus is a real Rust distinction:

`Copy` \iff `un` (fact) `owned non-Copy` \iff `affine` (resource)

And the **containment invariant** is enforced too: you **cannot** make a type `Copy` if it owns a `String` – a copyable type may not contain a moved (resource) field. Exactly `un` $\not\subseteq$ `lin`.

Since moving on every call would be painful, Rust supports **borrowing**, a relaxation of the affine discipline.

It **borrow**s a **value** using a **reference** (e.g., `&s`) to **use it without taking ownership**.

```
fn len(s: &String) -> usize { s.len() } // borrows, does not own
let s = String::from("hi");
let n = len(&s); // borrow: s is NOT moved
println!("{s}: {n}"); // OK -- we still own s
```

The **borrow checker** enforces scoped sharing rules:

- `&T` – a **shared** (read-only) borrow; many at once;
- `&mut T` – an **exclusive** (mutable) borrow; only one at a time.

The `&mut` rule is also key to preventing **data races** in concurrent code.

By **counting uses** at compile time, Rust gets **memory safety with no garbage collector**: no use-after-free, no double-free, no data races.

The **same** machinery powers many systems:

system	discipline	tracks
Rust	affine (move)	ownership; “use after move”
Linear Haskell	linear(a %1 → b)	“consumed exactly once”
Clean	uniqueness types	safe in-place update
session types	linear channels	protocol order (send/rcv)
quantum λ	linear	the no-cloning theorem

The key idea is to **treat a value as a resource** and let the type system account for every use.

- Effect Systems

Jihyeok Park
jihyeok_park@korea.ac.kr
<https://plrg.korea.ac.kr>