



# Практическая теория типов

Лекция 12: Substructural logics



# План лекции

1. Субструктурные логики и типы
2. Линейная логика и типы
3. Аффинная логика и типы
4. Примеры использования линейных и аффинных типов  
в Cyclone и Rust



# Субструктурная логика и типы

# Мотивация

- Естественный вывод представляет “естественный вывод” для рассуждений о выводимости:

$$A_1, \dots, A_n \vdash B$$

- Также, например, в пропозиционном исчислении можно записать следующее утверждение

$$p, \neg q \vdash q \Rightarrow (p \Rightarrow r)$$

- Согласно соотношению Карри-Ховарда, существует связь между логикой и типами: можно брать идеи из логики и использовать в языках программирования и наоборот.

# Мотивация

- Правила вывода, связанные с манипуляциями препосылками (формулами в контексте), называются субструктурными
- Они позволяют рассматривать контекст, как **множество** формул

$$\text{EXCHANGE} \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}$$

$$\text{CONTRACTION} \frac{\Gamma, A, A, \Delta \vdash B}{\Gamma, A, \Delta \vdash B}$$

$$\text{WEAKENING} \frac{\Gamma, \Delta \vdash B}{\Gamma, A, \Delta \vdash B}$$

- Если какая-то логическая система не использует одно из данных правил вывода, то она называется субструктурной

# Мотивация

- Контекст в логических системах соответствует состоянию программы (например, набору переменных и их значений) в определённый момент времени
- Используя идеи соответствия Карри-Ховарда, можно ограничивать операции над переменными, получая различные субструктурные системы типов:

	Exchange	Weakening	Contraction	Use
<b>Ordered</b>	—	—	—	Exactly once in order
<b>Linear</b>	Allowed	—	—	Exactly once
<b>Affine</b>	Allowed	Allowed	—	At most once
<b>Relevant</b>	Allowed	—	Allowed	At least once
<b>Normal</b>	Allowed	Allowed	Allowed	Arbitrarily



# Мотивация линейной логики

```
int main () {  
    char *ptr = malloc(sizeof(int));  
    *ptr = 3;  
    free(ptr);  
    return 0;  
}
```

- В языке C необходимо вручную деаллоцировать выделенную память, причём один раз
- Как построить систему типов, которая бы гарантировала это?



# Мотивация линейной логики

```
int main () {
```

```
    char *ptr = malloc(sizeof(int));
```

```
    *ptr = 3;
```

```
    return 0;
```

```
}
```

*Handwritten annotations:*

- ← *char \*ptr 1 = ptr;* (points to the `*ptr = 3;` line)
- free(ptr);*
- ptr = nullptr;*
- ptr 1* (points to the `ptr` variable in the `malloc` line)



$$\text{WEAKENING} \frac{\Gamma, \Delta \vdash B}{\Gamma, A, \Delta \vdash B}$$



# Мотивация линейной логики

```
int main () {
    char *ptr = malloc(sizeof(int));
    *ptr = 3;
    free(ptr);
    free(ptr);
    return 0;
}
```

$$\text{CONTRACTION} \frac{\Gamma, A, A, \Delta \vdash B}{\Gamma, A, \Delta \vdash B}$$



# Линейная логика

- В линейной логике не доступно ни ослабление, ни контракция
- Каждая переменная должна использоваться в точности один раз (нельзя копировать и не использовать)
- основано на идее ресурсов, которыми можно управлять и у которых есть владелец
- в основном используется для корректной работы с памятью, а также для корректной работы многопоточных программ
- линейная логика соответствует системе типов для process calculus, где
  - доказательства - процессы
  - утверждения - сессионные типы (коммуникационные протоколы)



# `std::unique_ptr`

К какому субструктурному типу относится `std::unique_ptr`?



# std::unique\_ptr

хотя std::unique\_ptr и использует некоторые идеи из линейной логики, он является нормальным типом с точки зрения субструктурной логики, потому что допускает множественное использование переменной:

```
#include <iostream>

struct C {};

int main()
{
    auto coin = std::make_unique<C>();
    std::cout << coin << std::endl;

    auto candy = std::move(coin);
    std::cout << coin << std::endl;

    auto drink = std::move(coin);
    std::cout << coin << std::endl;
}
```

Результат выполнения:

```
0x505338
0
0
```



# Линейная логика и типы



# Линейная логика: axiom

## Constructive Logic

$$\frac{A \in \Gamma}{\Gamma \vdash A} \text{ (HYP)}$$

## Linear Logic

$$\frac{}{A \vdash A} \text{ (HYP)}$$

Если дано только  $A$ , то можно использовать только его, не увеличивая контекст



# Линейная логика: импликация

- В классических логиках одной из основных конструкций была импликация  $A \rightarrow B$
- Её можно было интерпретировать, как то, что если истинно  $A$ , то истинно и  $B$ , или как про тип функции
- В классической логике же используется аналог, который означает, что если дано в точности одно  $A$ , можно использовать его, чтобы произвести в точности одно  $B$ :  $A \multimap B$
- можно думать о нём, как о функции, принимающей аргумент по значению, выводящей его из контекста и возвращающей одну копию  $B$

# Линейная логика: импликация

## Constructive Logic

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

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

## Linear Logic

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

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



# Линейная логика: импликация

## Vending machine

English	Logic	Rust
A coin can buy you a piece of candy, a drink, or go out of scope.	Coin $\multimap$ Candy Coin $\multimap$ Drink Coin $\multimap$ T	<pre>fn buy_candy(_ : Coin) -&gt; Candy { Candy{} } fn buy_drink(_ : Coin) -&gt; Drink { Drink{} }</pre>

# Линейная логика: конъюнкция

- Например, в классической и интуиционистской логике можно вывести

$$A \rightarrow B, A \rightarrow C, A \vdash B \wedge C$$

- но в линейной из-за правил вывода это не выводимо

$$A \multimap B, A \multimap C, A \not\vdash B \otimes C$$

- нужно иметь две копии значения  $A$

$$A \multimap B, A \multimap C, A, A \vdash B \otimes C$$

# Линейная логика: конъюнкция

## Constructive Logic

$$\begin{array}{c}
 \begin{array}{c} \mathcal{C} \quad \mathcal{C} \\ \Gamma \vdash A_1 \quad \Gamma \vdash A_2 \\ \hline \Gamma \vdash A_1 \wedge A_2 \end{array} (\wedge I) \\
 \swarrow \mathcal{C} \\
 \frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_1} (\wedge E1) \qquad \frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_2} (\wedge E2)
 \end{array}$$

## Linear Logic

$$\begin{array}{c}
 \begin{array}{c} \mathcal{C} \quad \mathcal{C} \\ \Delta_1 \vdash A_1 \quad \Delta_2 \vdash A_2 \\ \hline \Delta_1, \Delta_2 \vdash A_1 \otimes A_2 \end{array} (\otimes I) \\
 \swarrow \mathcal{C}, \mathcal{C} \\
 \frac{\Delta \vdash A_1 \otimes A_2 \quad \Delta', A_1, A_2 \vdash C}{\Delta, \Delta' \vdash C} (\otimes E)
 \end{array}$$

# Линейная логика: дизъюнкция

## Constructive Logic

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

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

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

## Linear Logic

$$\frac{\Delta \vdash A_1}{\Delta \vdash A_1 \oplus A_2} (\oplus I_1)$$

$$\frac{\Delta \vdash A_2}{\Delta \vdash A_1 \oplus A_2} (\oplus I_2)$$

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

# Линейные типы

- Линейные типы - это тоже самое для линейной логики, что и STLC для интуиционистской логики
- Основная идея линейных типов заключается в том, что для выполнения утверждения о типизации необходимо наличие в точности тех же самых переменных в контексте, что участвуют в утверждении о типизации  $\Gamma \vdash e : \tau$

# Линейные типы

- Грамматика для термов

$$e ::= x \mid \lambda x. e : \tau \mid e_1 e_2$$

- Грамматика для типов

$$\tau ::= b \mid \tau_1 \multimap \tau_2$$

- Правила типизации

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

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1. e : \tau_1 \multimap \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \multimap \tau_2 \quad \Delta \vdash e_2 : \tau_1}{\Gamma, \Delta \vdash e_1 e_2 : \tau_2}$$



# Линейные типы

В полученной системе типов не типизируется 1

$$\lambda f : \text{int} \multimap \text{int}. \lambda x : \text{int}. f (f x)$$

потому что  $f$  используется два раза



# Линейные типы и memory safety

- Посмотрим, как линейные типы могут предотвращать ошибки работы с памятью
- Введём операции malloc и free в STLC

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{malloc } e : \tau \text{ ptr}}$$

$$\frac{\Gamma \vdash e : \tau \text{ ptr}}{\Gamma \vdash \text{free } e : \text{unit}}$$

- Это позволит писать программы

```
let  $p$  = malloc 4 in  
free  $p$ ;  
free  $p$ 
```





# Линейные типы и memory safety

- Для того, чтобы использовать линейные типы, модифицируем malloc так, чтобы он возвращал не только указатель, но и сар
- Модифицируем free таким образом, чтобы функция принимала не только указатель, но и сар
- Применение рассмотренных линейных типов для сар можно гарантировать, что программа удалит указатель в точности один раз
- сар можно передать в другую функцию, которая не вызывает внутри free, если она возвращает его обратно в качестве результата
- Данная идея была использована в языке Cyclone и является ключевой для обработки lifetime языка Rust
- Больше подробностей в статье “Linear regions are all you need”

# Линейные типы и memory safety

```
region h {
    int*h x = rmalloc(h, sizeof(int));
    int?h y = rnew(h) { 1, 2, 3 };
    char?h z = rprintf(h, "hello");
}
```

*Pointer types are annotated with the name of the region 'h'*

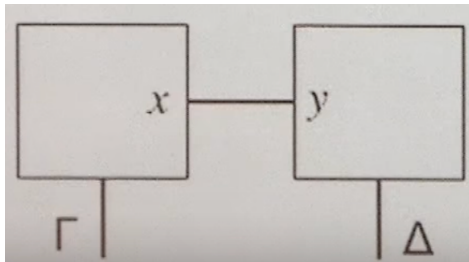


# Что такое дедлок

Дедлок возникает из-за циклической зависимости между коммуникационными операциями, когда два процесса имеют несколько каналов

input on a then output on b		output on a then input on b	OK
input on a then output on b		input on b then output on a	STUCK

Линейная система типов гарантирует отсутствие дедлоков, потому что гарантирует, что процессы связаны только одним каналом



$$\begin{array}{c}
 (\text{cut}) \\
 \frac{P \vdash \Gamma, x:A \quad Q \vdash \Delta, y:A^\perp}{(\nu x^A y)(P \mid Q) \vdash \Gamma, \Delta}
 \end{array}$$



# Аффинная логика и типы/Rust



# Аффинные типы

Аффинные типы гарантируют, что переменная будет использоваться не более одного раза

```
val f = openFile "free_uc_stones.gif"  
val () = closeFile f  
val () = closeFile f
```

```
val f = openFile "free_uc_stones.gif"
```

# Rust

В основе Rust лежит концепция владения и заимствования

```
fn main() {  
    let owner: Vec<i32> = vec![1, 2, 3];  
}
```

→ 'owner' owns the vector



# Rust

В основе Rust лежит концепция владения и заимствования

```
fn main() {  
    let owner: Vec<i32> = vec![1, 2, 3];  
    let reference: &Vec<i32> = &owner;  
}
```

*let owner\_1 = owner;*

*'reference' borrows 'owner'*



# Rust

Также используется концепция мутабельности типов

```
fn main() {
    let mut owner: Vec<i32> = vec![1, 2, 3];
    owner.push(4);

    let reference: &mut Vec<i32> = &mut owner;
    reference.push(5);
}
```





# Rust



Существует три правила владения, которые проверяется borrow-checker:

- все значения должны иметь в точности одного владельца
- ссылка не может иметь лайфтайм больше, чем у владельца
- на одно значение может существовать или одна мутабельная ссылка, или много иммутабельных



# Rust

- все значения должны иметь в точности одного владельца

```
fn main() {
    let v = vec![1, 2, 3];
    let v2 = v;
    print(v);
}
```

*let v3 = &mut v;*



# Rust

- все значения должны иметь в точности одного владельца

```
fn main() {
    let v = vec![1, 2, 3];
    let v2 = v;
    print(v);
}
```

*let v = ...;*

value moved here

value used here after move



# Rust

- ссылка не может иметь лайфтайм больше, чем у владельца

```
fn main() {
    let v = vec![1, 2, 3];
    let x = &v[0];
    let v2 = v;
    let y = x + 1;
}
```



# Rust

- ссылка не может иметь лайфтайм больше, чем у владельца

```
fn main() {
    let v = vec![1, 2, 3];
    let x = &v[0];
    let v2 = v;
    let y = x + 1;
}
```

Diagram illustrating memory movement and borrowing:

- A blue box highlights the variable `v` and its usage in `&v[0]` and `v2 = v`.
- A red line labeled *moved* points from `v` to `v2`, indicating that `v` is moved to `v2`.
- A blue line labeled *reference* points from `x` to `v`, indicating that `x` is a reference to `v`.

unsound bugs  
unsafe

```
fn main() {
    let v = vec![1, 2, 3];
    let x = &v[0];
    let v2 = v;
    let y = x + 1;
}
```

Annotations for the second code block:

- `&v[0]`: borrow out of ``v`` occurs here
- `v`: move out of ``v`` occurs here
- `x + 1`: borrow later used here



# Rust

- на одно значение может существовать или одна мутабельная ссылка, или много иммутабельных

```
fn main() {
    let mut v = vec![1, 2, 3];
    let x = &v[0];
    v.push(4);
    let y = x + 1;
}
```

*push(2 mut self,*



# Rust

- на одно значение может существовать или одна мутабельная ссылка, или много иммутабельных

```
fn main() {
    let mut v = vec![1, 2, 3];
    let x = &v[0];
    v.push(4);
    let y = x + 1;
}
```

Diagram illustrating memory mutability:

- A blue box highlights the `v.push(4);` line, labeled *mutable* in red.
- A purple box highlights the `let y = x + 1;` line, labeled *immutable* in blue.
- A blue line connects the `x` variable to the `v.push(4);` line.
- A blue line connects the `x` variable to the `let y = x + 1;` line.

X

```
fn push(&mut self, value: i32) { /* ... */ }
```



# Rust

- на одно значение может существовать или одна мутабельная ссылка, или много иммутабельных

```
fn main() {
    let mut v = vec![1, 2, 3];
    let x = &v[0];           immutable borrow occurs here
    v.push(4);               mutable borrow occurs here
    let y = x + 1;           immutable borrow later used here
}
```





# Rust: лайфтаймы

2. A reference to a value cannot outlive the owner.

```
fn main() {
    let mut v = vec![1, 2, 3];
    let x = &v[0];
    v.push(4);
    let y = x + 1;
}
```

```
fn main() {
    let mut v = vec![1, 2, 3];
    let x = &v[0];
    v.push(4);
    let y = x + 1;
}
```

Diagram illustrating the lifetime of variables in the code snippet above:

- The variable `v` is the owner of the memory for the vector.
- The variable `x` is a reference to the first element of the vector, `v[0]`.
- The variable `y` is a reference to the first element of the vector, `v[0]`.
- The variable `a` is a reference to the first element of the vector, `v[0]`.

