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:

$$A_1, \ldots, A_n \vdash B$$

- 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

$$\textsc{Exchange}\ \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C} \qquad \textsc{Contraction}\ \frac{\Gamma, A, A, \Delta \vdash B}{\Gamma, A, \Delta \vdash B} \qquad \textsc{Weakening}\ \frac{\Gamma, \Delta \vdash B}{\Gamma, A, \Delta \vdash B}$$

- 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
Ordered	—	—	—	Exactly once in order
Linear	Allowed	—	—	Exactly once
Affine	Allowed	Allowed	—	At most once
Relevant	Allowed	—	Allowed	At least once
Normal	Allowed	Allowed	Allowed	Arbitrarily

Мотивация линейной логики


```c
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;
}
```

$$\textsc{Weakening}\ \frac{\Gamma, \Delta \vdash B}{\Gamma, A, \Delta \vdash B}$$

Мотивация линейной логики

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

$$\textsc{Contraction}\ \frac{\Gamma, A, A, \Delta \vdash B}{\Gamma, A, \Delta \vdash B}$$

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

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

Linear Logic

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

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

Линейная логика: импликация

- В классических логиках одной из основных конструкций была импликация $A \to B$
- 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

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

Linear Logic

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

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 ⊸ Candy Coin ⊸ Drink Coin ⊸ ⊤	`fn buy_candy(_: Coin) -> Candy { Candy{} }` `fn buy_drink(_: Coin) -> Drink { Drink{} }`

Линейная логика: конъюнкция

- Например, в классической и интуиционисткой логике можно вывести

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

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

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

- нужно иметь две копии значения A

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

Линейная логика: конъюнкция

Constructive Logic

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

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

Linear Logic

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

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

Линейная логика: дизъюнкция

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

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

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

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

$$\frac{}{x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\, e : \tau_1 \multimap \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Delta \vdash e_2 : \tau_1}{\Gamma, \Delta \vdash e_1\ e_2 : \tau_2}$$

Линейные типы

В полученной системе типов не типизируется 1

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

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

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{malloc}\ e : \tau\ \mathsf{ptr}} \qquad \frac{\Gamma \vdash e : \tau\ \mathsf{ptr}}{\Gamma \vdash \mathsf{free}\ e : \mathsf{unit}}$$

- Это позволит писать программы

$$\begin{array}{l} \mathsf{let}\ p\ =\ \mathsf{malloc}\ 4\ \mathsf{in} \\ \mathsf{free}\ p; \\ \mathsf{free}\ p \end{array}$$

Линейные типы и memory safety

- Для того, чтобы использовать линейные типы, модифицируем malloc так, чтобы он возвращал не только указатель, но и cap
- Модифицируем free таким образом, чтобы функция принимала не только указатель, но и cap
- Применение рассмотренных линейных типов для cap можно гарантировать, что программа удалит указатель в точности один раз
- cap 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");
}
```

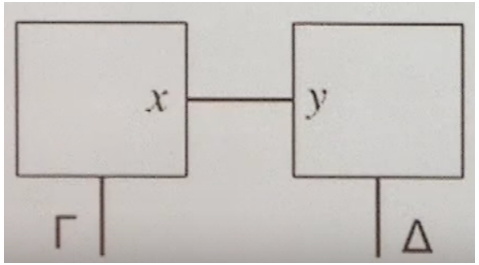
Pointer types are annotated with the name of the region 'h

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 гарантирует, что процессы связаны только одним каналом



$$\frac{P \vdash \Gamma, x{:}A \qquad Q \vdash \Delta, y{:}A^{\perp}}{(\nu x^{A} y)(P \mid Q) \vdash \Gamma, \Delta}\ (\text{cut})$$

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

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

Rust

- все значения должны иметь в точности одного владельца

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

moved

reference

```
fn main() {
  let v = vec![1, 2, 3];
  let x = &v[0];          borrow out of `v` occurs here
  let v2 = v;             move out of `v` occurs here
  let y = 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;
}
```

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;
}



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;
}
```