RustCon 2025

Автоматическое дифференцирование в Rust

Кравченко Владимир

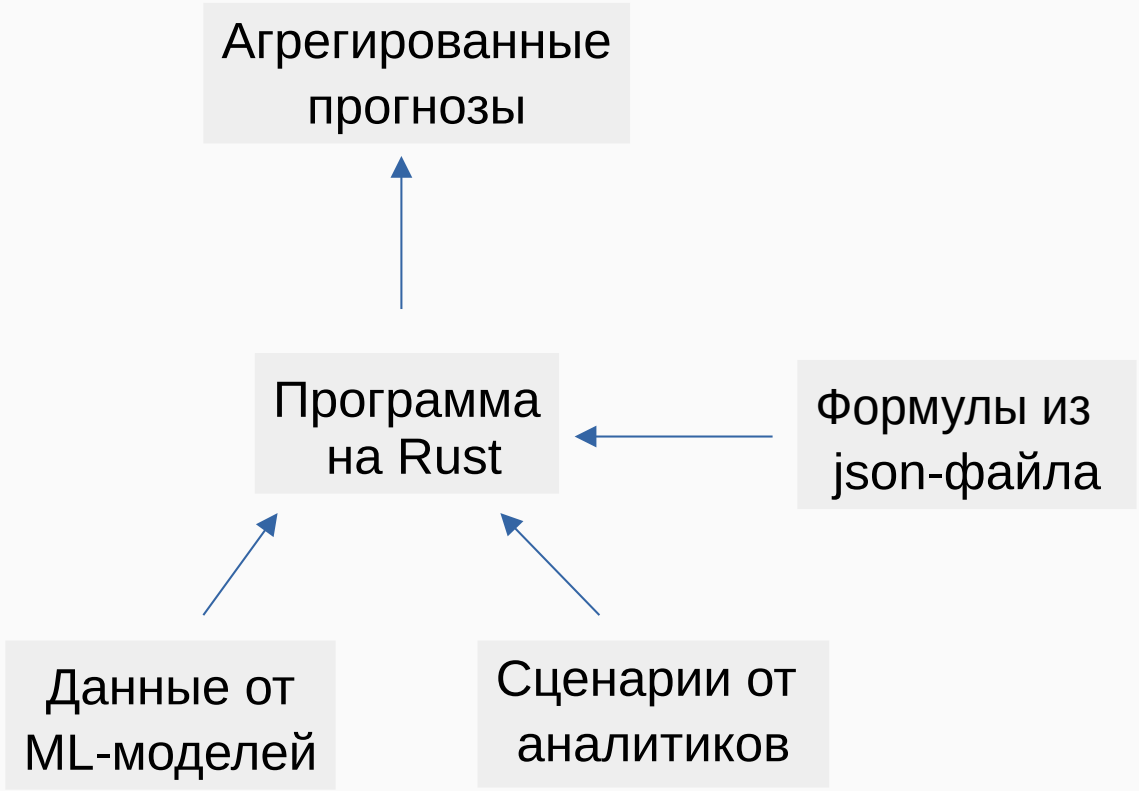
Обо мне

Python в области научных вычислений свыше 5 лет.

Последние 3 года компилирую Rust.

Область интересов: численные методы и машинное обучение.

Попытка применить автодиф в проде



Rust Project goals

2024h2 - Expose experimental LLVM features for automatic differentiation and GPU offloading

2025h2 - Finish the `std::offload` module

- `std::batching`
- **`std::autodiff`**
- `std::offload`

Что такое дифференцирование?

Дихотомия подходов

Полное знание — Чёрный ящик

Аналитический подход — Конечная разность

Аналитический подход. Элементарные функции

Элементарные функции:

$$\frac{\mathrm{d}}{\mathrm{d}x} c = 0$$

$$\frac{\mathrm{d}}{\mathrm{d}x} x^n = n x^{n-1}$$

$$\frac{\mathrm{d}}{\mathrm{d}x} \sin(x) = \cos x$$

Векторно-матричные:

$$\nabla_{\{\boldsymbol{x}\}} (\boldsymbol{x}^{\mathrm{T}} \boldsymbol{A} \boldsymbol{x}) = (\boldsymbol{A} + \boldsymbol{A}^{\mathrm{T}}) \boldsymbol{x}$$

Правила:

$$(fg)'(x) = f'(x)g(x) + f(x)g'(x)$$

$$\frac{\mathrm{d}}{\mathrm{d}x} f(g(x)) = f'(g(x)) \cdot g'(x)$$

Аналитический подход: SymPy ⟶ Rust

Шаг 1

$$f(\boldsymbol{x}) = \boldsymbol{x}^{\mathrm{T}} \boldsymbol{A} \boldsymbol{x}$$

```
A = MatrixSymbol("A", n, n)  # матрица
x = MatrixSymbol("x", n, 1)  # вектор

f = x.T * A * x # наша функция

grad = derive_by_array(f, x) # градиент
grad = simplify(grad) # пытаемся упростить
print(grad)

>> "A*x + A.T*x"
```

Аналитический подход: SymPy → Rust

Шаг 1

$$f(\boldsymbol{x}) = \boldsymbol{x}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{x}$$

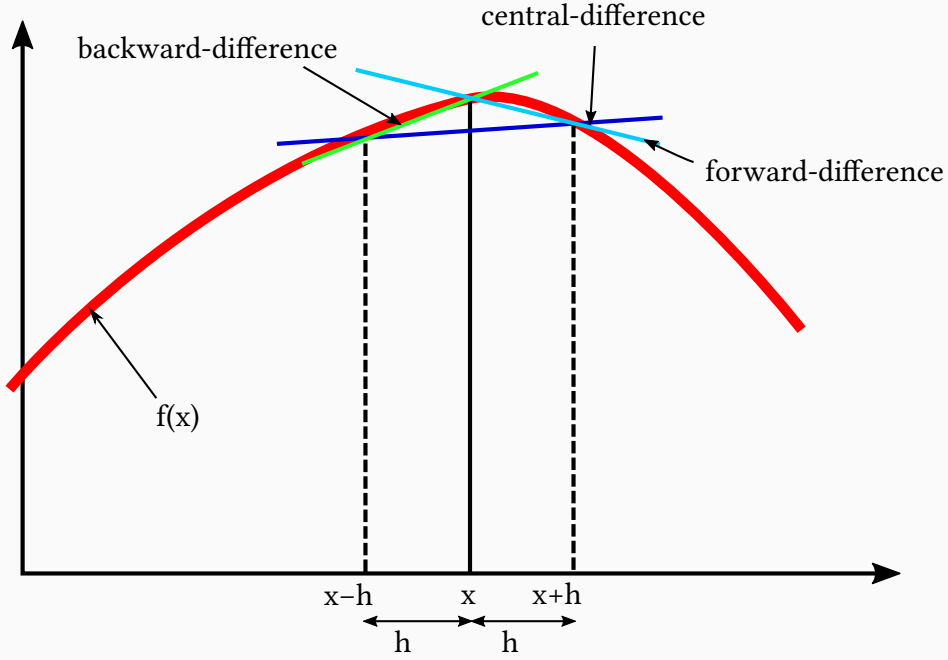
```python
A = MatrixSymbol("A", n, n)  # матрица
x = MatrixSymbol("x", n, 1)  # вектор

f = x.T * A * x # наша функция

grad = derive_by_array(f, x) # градиент
grad = simplify(grad) # пытаемся упростить
print(grad)

>> "A*x + A.T*x"
```

Шаг 2

$$\nabla_{\{\boldsymbol{x}\}} f(\boldsymbol{x}) = (\boldsymbol{A} + \boldsymbol{A}^{\mathrm{T}})\boldsymbol{x}$$

```rust
pub fn f(
    a: ArrayView2<'_, f64>,
    x: ArrayView1<'_, f64>
) -> Array1<f64> {

    (&a + &a.t()).dot(&x)
}
```

Конечные разности



Прямая разность (сложность $n + 1$):

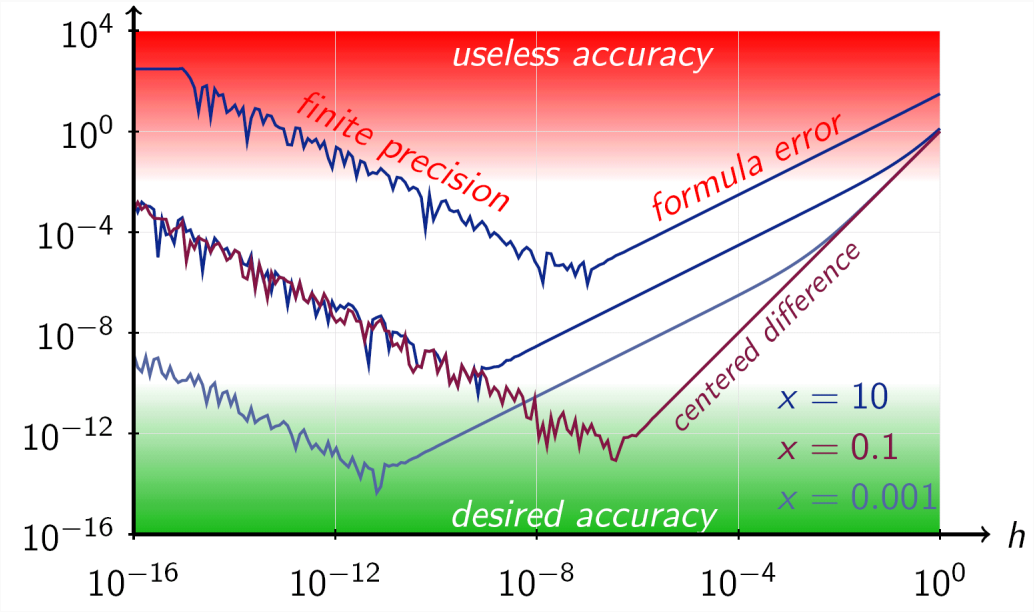
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Центральная разность (сложность $2n$):

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Конечные разности (минусы)

Точность падает в уменьшением h



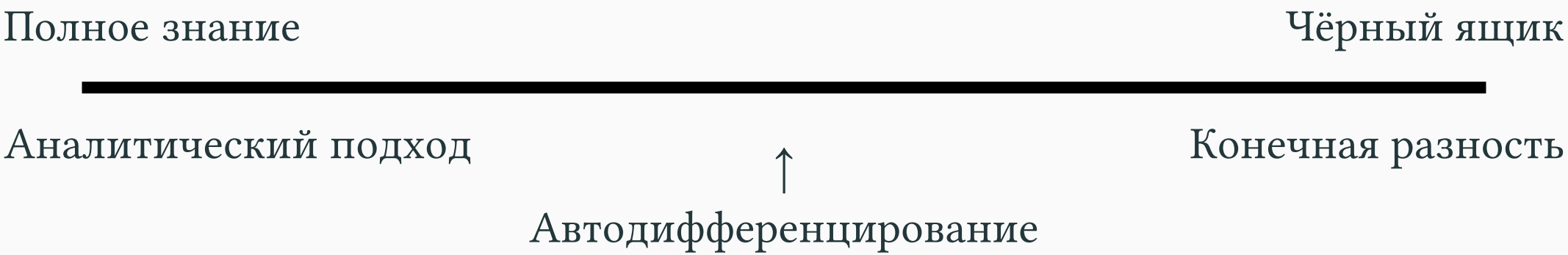
Прямая разность (точность $O(h)$):

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Центральная разность (точность $O(h^2)$):

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Автодифференцирование



1. Работает с вашим кодом - не требует доп. формализации
2. Машинная точность производной
3. Стоимость как несколько вызовов исходной функции

Автодифференцирование

Режимы автоматического дифференцирования

- **Прямой режим** (forward mode, JVP — Jacobian–Vector Product)
- **Обратный режим** (reverse mode, VJP — Vector–Jacobian Product)

Forward mode: Арифметика дуальных чисел

Обозначения:

$$x = x_0 + x_1\varepsilon, \qquad y = y_0 + y_1\varepsilon, \qquad \varepsilon^2 = 0$$

Forward mode: Арифметика дуальных чисел

Обозначения:

$$x = x_0 + x_1\varepsilon, \qquad y = y_0 + y_1\varepsilon, \qquad \varepsilon^2 = 0$$

Сложение:

$$(x_0 + x_1\varepsilon) + (y_0 + y_1\varepsilon) = (x_0 + y_0) + (x_1 + y_1)\varepsilon$$

Forward mode: Арифметика дуальных чисел

Обозначения:

$$x = x_0 + x_1\varepsilon, \qquad y = y_0 + y_1\varepsilon, \qquad \varepsilon^2 = 0$$

Сложение:

$$(x_0 + x_1\varepsilon) + (y_0 + y_1\varepsilon) = (x_0 + y_0) + (x_1 + y_1)\varepsilon$$

Умножение:

$$\begin{aligned}(x_0 + x_1\varepsilon) \cdot (y_0 + y_1\varepsilon) = \\ = x_0y_0 + x_0y_1\varepsilon + x_1y_0\varepsilon + x_1y_1\varepsilon^2 = \\ = (x_0 \cdot y_0) + (x_0 \cdot y_1 + x_1 \cdot y_0)\varepsilon\end{aligned}$$

Forward mode: Арифметика дуальных чисел

Обозначения:

$$x = x_0 + x_1\varepsilon, \qquad y = y_0 + y_1\varepsilon, \qquad \varepsilon^2 = 0$$

Сложение:

$$(x_0 + x_1\varepsilon) + (y_0 + y_1\varepsilon) = (x_0 + y_0) + (x_1 + y_1)\varepsilon$$

Умножение:

$$\begin{aligned}(x_0 + x_1\varepsilon) \cdot (y_0 + y_1\varepsilon) = \\ = x_0y_0 + x_0y_1\varepsilon + x_1y_0\varepsilon + \cancel{x_1y_1\varepsilon^2} = \\ = (x_0 \cdot y_0) + (x_0 \cdot y_1 + x_1 \cdot y_0)\varepsilon\end{aligned}$$

Forward mode: Арифметика дуальных чисел

Обозначения:

$$x = x_0 + x_1\varepsilon, \qquad y = y_0 + y_1\varepsilon, \qquad \varepsilon^2 = 0$$

Сложение:

$$(x_0 + x_1\varepsilon) + (y_0 + y_1\varepsilon) = (x_0 + y_0) + (x_1 + y_1)\varepsilon$$

Умножение:

$$\begin{aligned}(x_0 + x_1\varepsilon) \cdot (y_0 + y_1\varepsilon) = \\ &= x_0y_0 + x_0y_1\varepsilon + x_1y_0\varepsilon + = \\ &= (x_0 \cdot y_0) + (\cancel{x_0 \cdot y_1} + x_1 \cdot y_0)\varepsilon\end{aligned}$$

Forward mode: Арифметика дуальных чисел

Обозначения:

$$x = x_0 + x_1\varepsilon, \qquad y = y_0 + y_1\varepsilon, \qquad \varepsilon^2 = 0$$

Правило для функций:

$$f(x_0 + x_1\varepsilon) = f(x_0) + f'(x_0)x_1\varepsilon$$

Forward mode: Арифметика дуальных чисел

Обозначения:

$$x = x_0 + x_1\varepsilon, \qquad y = y_0 + y_1\varepsilon, \qquad \varepsilon^2 = 0$$

Правило для функций:

$$f(x_0 + x_1\varepsilon) = f(x_0) + f'(x_0)x_1\varepsilon$$

Примеры:

$$\sin(x) = \sin(x_0) + \cos(x_0)x_1\varepsilon$$

$$\log(x) = \log(x_0) + \left(\frac{x_1}{x_0}\right)\varepsilon$$

Forward mode: Реализация на Rust

```rust
struct Dual {
  p: f64,  // Значение ("primal")
  d: f64,  // Производная ("tangent")
}
```

```rust
fn add(self, rhs: Dual) → Dual;
fn sub(self, rhs: Dual) → Dual;
fn neg(self) → Dual;
fn mul(self, rhs: Dual) → Dual;
fn div(self, rhs: Dual) → Dual;
fn powf(self, k: f64) → Dual;
fn powi(self, k: i32) → Dual;
fn sin(self) → Dual;
fn cos(self) → Dual;
fn exp(self) → Dual;
fn ln (self) → Dual;
```

...

Forward mode: Реализация на Rust

$$f(x, y) = x \cdot y + \sin(x) \cdot y$$

```
fn f(x:Dual, y:Dual) → Dual  {
  x * y + x.sin() * y
}
```

Forward mode: Пример вызова

Прямой проход - выставляем 1 там где надо, а где не надо выставляем 0.

```
fn f(x:Dual, y:Dual) → Dual;

let (x, y) = (1.2, 3.4);

let (v, dx) = f(
  Dual { p: x, d: 1.0 },  // ∂f/∂x: берём по x
  Dual { p: y, d: 0.0 },  // держим y константой
);

let (v, dy) = f(
  Dual { p: x, d: 0.0 },  // держим x константой
  Dual { p: y, d: 1.0 },  // ∂f/∂y: берём по y
);
```

Forward mode: Библиотеки

На уровне исходного кода:

- **num-dual** — обобщённые (hyper-)dual числа
- **autodiff** — форвард-режим через dual числа (diff/grad)
- **numdiff** — форвард-AD на dual числах плюс численная разность

Reverse mode

- **Обратный режим** (reverse mode, VJP — Vector–Jacobian Product)

Reverse mode: Определение

$$f(x, y) = x \cdot y + \sin(x) \cdot y$$

```rust
fn f(x: f64, y: f64) -> f64 {
    x * y + x.sin() * y
}
```

↓

```rust
/// Некоторая структура
struct Tape { /* ... */ }

///  Прямой проход - расчёт значения функции и некого `Tape`
fn f_augmented_primal(x: f64, y: f64) -> (f64, Tape);

/// Обратный проход - получение дифференциала (dx, dy)
fn f_reverse(x: f64, y: f64, seed_df: f64, tape: &Tape) -> (f64, f64);
```

Reverse mode: Прямой проход

```
fn f(x: f64, y: f64) → f64 {
    x * y + x.sin() * y
}
```

```
fn f_augmented_primal(x: f64, y: f64) → (f64, Tape) {
    let s1 = x * y;
    let sinx = x.sin();
    let s2 = sinx * y;
    let f = s1 + s2;

    (f, Tape { sinx })
}

struct Tape { sinx: f64 }
```

Reverse mode: Обратный проход

```rust
struct Tape { sinx: f64 }

fn f_augmented_primal(x: f64, y:
f64) → (f64, Tape) {
    let s1 = x * y;
    let sinx = x.sin();
    let s2 = sinx * y;
    let f = s1 + s2;

    (f, Tape { sinx })
}
```

```rust
fn f_reverse(x: f64, y: f64, seed_df: f64, tape:
&Tape) → (f64, f64) {
    let (mut bx, mut by) = (0.0, 0.0);
    /* ... */
```

Reverse mode: Обратный проход

```rust
struct Tape { sinx: f64 }

fn f_augmented_primal(x: f64, y:
f64) → (f64, Tape) {
    let s1 = x * y;
    let sinx = x.sin();
    let s2 = sinx * y;
    let f = s1 + s2;

    (f, Tape { sinx })
}
```

```rust
fn f_reverse(x: f64, y: f64, seed_df: f64, tape:
&Tape) → (f64, f64) {
    let (mut bx, mut by) = (0.0, 0.0);

    let bs1 = 1.0 * seed_df;   ∂f/∂s1 = 1

    /*  ...  */
```

Reverse mode: Обратный проход

```rust
struct Tape { sinx: f64 }

fn f_augmented_primal(x: f64, y:
f64) → (f64, Tape) {
    let s1 = x * y;
    let sinx = x.sin();
    let s2 = sinx * y;
    let f = s1 + s2;

    (f, Tape { sinx })
}
```

```rust
fn f_reverse(x: f64, y: f64, seed_df: f64, tape:
&Tape) → (f64, f64) {
    let (mut bx, mut by) = (0.0, 0.0);

    let bs1 = 1.0 * seed_df;
    let bs2 = 1.0 * seed_df;   ∂f/∂s2 = 1
    /* ... */
```

Reverse mode: Обратный проход

```rust
struct Tape { sinx: f64 }

fn f_augmented_primal(x: f64, y:
f64) → (f64, Tape) {
    let s1 = x * y;
    let sinx = x.sin();
    let s2 = sinx * y;
    let f = s1 + s2;

    (f, Tape { sinx })
}
```

```rust
fn f_reverse(x: f64, y: f64, seed_df: f64, tape:
&Tape) → (f64, f64) {
    let (mut bx, mut by) = (0.0, 0.0);

    let bs1 = 1.0 * seed_df;
    let bs2 = 1.0 * seed_df;

    let bsinx = y * bs2;   ∂s2/∂sinx = y

    /* ... */
```

Reverse mode: Обратный проход

```rust
struct Tape { sinx: f64 }

fn f_augmented_primal(x: f64, y:
f64) → (f64, Tape) {
    let s1 = x * y;
    let sinx = x.sin();
    let s2 = sinx * y;
    let f = s1 + s2;

    (f, Tape { sinx })
}
```

```rust
fn f_reverse(x: f64, y: f64, seed_df: f64, tape:
&Tape) → (f64, f64) {
    let (mut bx, mut by) = (0.0, 0.0);

    let bs1 = 1.0 * seed_df;
    let bs2 = 1.0 * seed_df;

    let bsinx = y * bs2;
    by += tape.sinx * bs2;   ∂s2/∂y = sinx
    /* ... */
```

Reverse mode: Обратный проход

```
struct Tape { sinx: f64 }

fn f_augmented_primal(x: f64, y:
f64) → (f64, Tape) {
    let s1 = x * y;
    let sinx = x.sin();
    let s2 = sinx * y;
    let f = s1 + s2;

    (f, Tape { sinx })
}
```

```
fn f_reverse(x: f64, y: f64, seed_df: f64, tape:
&Tape) → (f64, f64) {
    let (mut bx, mut by) = (0.0, 0.0);

    let bs1 = 1.0 * seed_df;
    let bs2 = 1.0 * seed_df;

    let bsinx = y * bs2;
    by += tape.sinx * bs2;
    bx += x.cos() * bsinx;   ∂sinx/∂x = cos(x)
    /* ... */
```

Reverse mode: Обратный проход

```
struct Tape { sinx: f64 }

fn f_augmented_primal(x: f64, y:
f64) → (f64, Tape) {
    let s1 = x * y;
    let sinx = x.sin();
    let s2 = sinx * y;
    let f = s1 + s2;

    (f, Tape { sinx })
}
```

```
fn f_reverse(x: f64, y: f64, seed_df: f64, tape:
&Tape) → (f64, f64) {
    let (mut bx, mut by) = (0.0, 0.0);

    let bs1 = 1.0 * seed_df;
    let bs2 = 1.0 * seed_df;

    let bsinx = y * bs2;
    by += tape.sinx * bs2;
    bx += x.cos() * bsinx;
    bx += y * bs1;      ∂s1/∂x = y
    by += x * bs1;      ∂s1/∂y = x

    (bx, by)
}
```

Reverse mode: Пример вызова

```
let (x, y) = (1.2, 3.4);
let (f_val, tape) = f_augmented_primal(x, y);
let (dx, dy) = f_reverse(x, y, /* seed_df */ 1.0, &tape);
```

Reverse mode: Библиотеки

Динамический DAG в памяти:

- **Burn** (burn-autodiff) — декоратор бэкенда с динамическим графом.
- **Candle** (burn-autodiff) — динамический граф вычислений и backprop.
- **autograd** (rust-autograd) — define-by-run, ленивое вычисление графа.
- **reverse** — явный tape для backprop.
- **dfdx** — DL-библиотека с backprop и автодифом

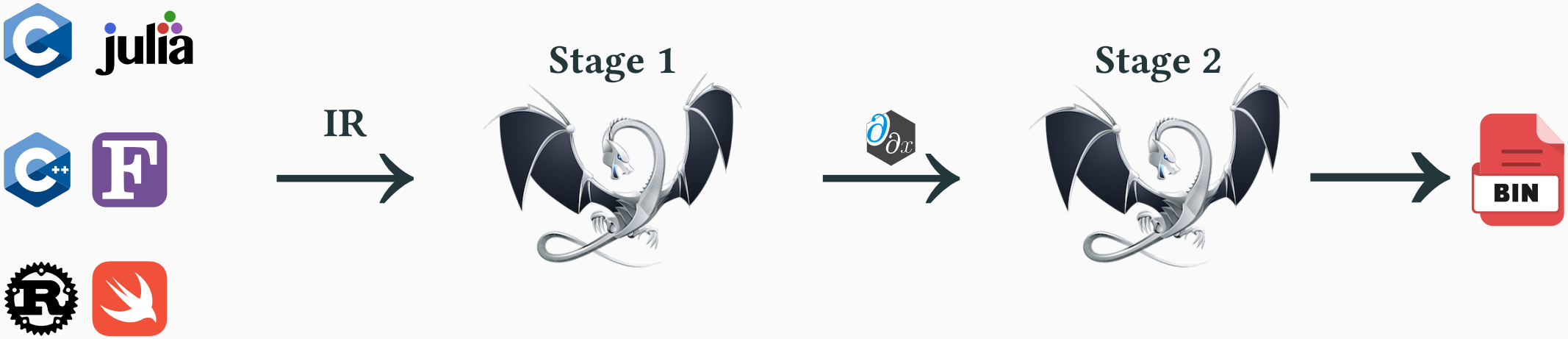
Какой алгоритм выбрать?

Смотрим на размерность входных и выходных данных

Подход	Стоимость на $\nabla f(\boldsymbol{x})$
Автодифф: прямой режим	$\sim n \times \mathrm{cost}(f)$ (один проход на вход)
Автодифф: обратный режим	$\sim 2..4 \times \mathrm{cost}(f)$ (на скалярный выход)

std::autodiff (Enzyme AD)

Enzyme AD



Stage 1 ≈ O2 без loop-vectorize, slp-vectorize, loop-unroll

Установка

```
# Выкачиваем репу
git clone https://github.com/rust-lang/rust.git
cd rust
```

Установка

```
# Выкачиваем репу
git clone https://github.com/rust-lang/rust.git
cd rust

# Выставляем необходимые флаги компиляции
./configure --enable-llvm-link-shared --enable-llvm-plugins --enable-llvm-enzyme --release-channel=nightly --enable-llvm-assertions --enable-clang --enable-lld --enable-option-checking --enable-ninja --disable-docs

# Собирае компилятор
./x build --stage 1 library
```

Установка

```
# Выкачиваем репу
git clone https://github.com/rust-lang/rust.git
cd rust

# Выставляем необходимые флаги компиляции
./configure --enable-llvm-link-shared --enable-llvm-plugins --enable-llvm-enzyme --release-
channel=nightly --enable-llvm-assertions --enable-clang --enable-lld --enable-option-
checking --enable-ninja --disable-docs

# Собирае компилятор
./x build --stage 1 library

# Добавляем в тулчейн
rustup toolchain link enzyme build/host/stage1
rustup toolchain install nightly
```

https://enzyme.mit.edu/rust/installation.html

Компиляция

Собирается программа так:

```
RUSTFLAGS="-Z autodiff=Enable" cargo +enzyme build --release
```

Компиляция

Собирается программа так:

```
RUSTFLAGS="-Z autodiff=Enable" cargo +enzyme build --release
```

Без LTO пока не собирается:

```
[profile.release]
lto = "fat"
```

Без `-Cembed-bitcode=yes` Enzyme не видит биткод зависимостей на стадии LTO и `rustc` падает с ошибкой `Can't find section .llvmbc`.

Just команды

Скачать и скомпилировать rustc+enzyme:

```
just install
```

Запустить какой-то из примеров из директории `crates`:

```
just run <crate_name>
```

Посмотреть декларацию дифф. функции (развернуть до AST):

```
just expand <crate_name>
```

std::autodiff

```
#[autodiff_forward(NAME, INPUT_ACTIVITIES, OUTPUT_ACTIVITY)]
#[autodiff_reverse(NAME, INPUT_ACTIVITIES, OUTPUT_ACTIVITY)]


#[autodiff_reverse(
    df,                     // имя сгенерированной функции
    Active, Active,         // активности аргументов (x, y)
    Active,                 // активен скалярный результат f
)]
fn f(x: f64, y: f64) → f64 {
  /*  ...  */
}
```

std::autodiff

```
#[autodiff_forward(NAME, INPUT_ACTIVITIES, OUTPUT_ACTIVITY)]
#[autodiff_reverse(NAME, INPUT_ACTIVITIES, OUTPUT_ACTIVITY)]


#[autodiff_reverse(
    df,                     // имя сгенерированной функции
    Active, Active,         // активности аргументов (x, y)
    Active,                 // активен скалярный результат f
)]
fn f(x: f64, y: f64) -> f64 {
  /* ... */
}
```

std::autodiff

```
#[autodiff_forward(NAME, INPUT_ACTIVITIES, OUTPUT_ACTIVITY)]
#[autodiff_reverse(NAME, INPUT_ACTIVITIES, OUTPUT_ACTIVITY)]


#[autodiff_reverse(
    df,                     // имя сгенерированной функции
    Active, Active,          // активности аргументов (x, y)
    Active,                 // активен скалярный результат f
)]
fn f(x: f64, y: f64) -> f64 {
  /*  ...  */
}
```

std::autodiff

```
#[autodiff_forward(NAME, INPUT_ACTIVITIES, OUTPUT_ACTIVITY)]
#[autodiff_reverse(NAME, INPUT_ACTIVITIES, OUTPUT_ACTIVITY)]


#[autodiff_reverse(
    df,                     // имя сгенерированной функции
    Active, Active,         // активности аргументов (x, y)
    Active,                  // активен скалярный результат f
)]
fn f(x: f64, y: f64) -> f64 {
  /* ... */
}
```

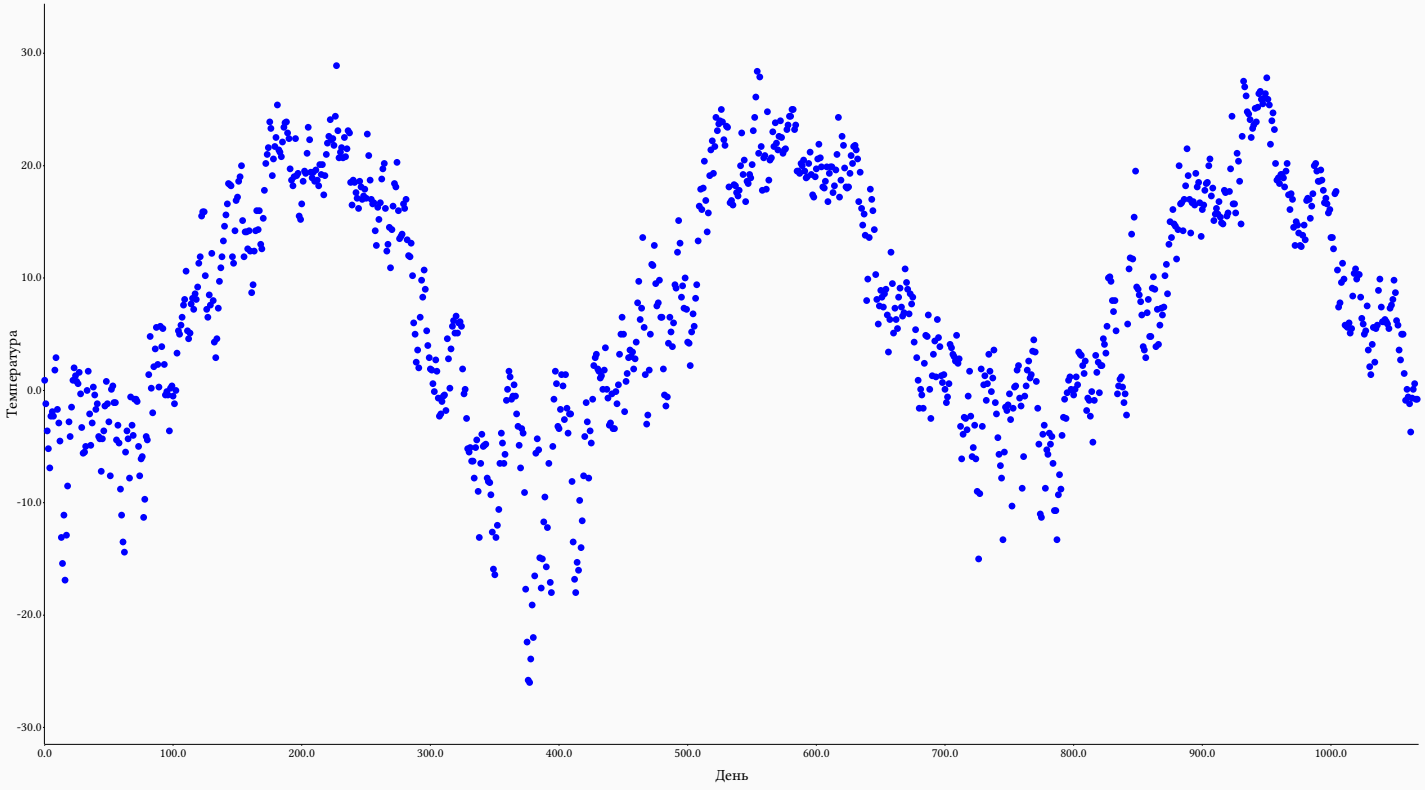
std::autodiff

```
#[autodiff_forward(NAME, INPUT_ACTIVITIES, OUTPUT_ACTIVITY)]
#[autodiff_reverse(NAME, INPUT_ACTIVITIES, OUTPUT_ACTIVITY)]

enum DiffActivity {
    None,           // Не участвует в дифференцировании
    Const,          // Учитывается как константа
    Active,         // (RM) Для скаляров f32, f64
    ActiveOnly,     // (RM) Считаем только градиент, без исходного значения
    Duplicated,     // (RM) для &T или *T, предоставляем свой теневой буфер
    DuplicatedOnly, // (RM) То же, но без пересчёта исходного значения (только градиент)
    Dual,           // (FM) Значение с одной "тенью"
    Dualv,          // (FM) Значение с векторизованной "тенью" (несколько направлений)
    DualOnly,       // (FM) Только "тень", без пересчёта значения
    DualvOnly,      // (FM) Только векторизованные "тени", без значения
}
```

std::autodiff

```
#[autodiff_forward(NAME, INPUT_ACTIVITIES, OUTPUT_ACTIVITY)]
#[autodiff_reverse(NAME, INPUT_ACTIVITIES, OUTPUT_ACTIVITY)]

enum DiffActivity {
    None,           // Не участвует в дифференцировании
    Const,          // Учитывается как константа
    Active,         // (RM) Для скаляров f32, f64
    ActiveOnly,     // (RM) Считаем только градиент, без исходного значения
    Duplicated,     // (RM) для &T или *T, предоставляем свой теневой буфер
    DuplicatedOnly, // (RM) То же, но без пересчёта исходного значения (только градиент)
    Dual,           // (FM) Значение с одной "тенью"
    Dualv,          // (FM) Значение с векторизованной "тенью" (несколько направлений)
    DualOnly,       // (FM) Только "тень", без пересчёта значения
    DualvOnly,      // (FM) Только векторизованные "тени", без значения
}
```

Пример обратного прохода



Суточная температура по СПб.

Пример обратного прохода

Модель: $t_i = a \cdot \sin\left(2\pi \frac{i+c}{p}\right) + b$

```
fn model(i: f32, a: f32, b: f32, p: f32, c: f32) → f32 {
    let arg = 2.0 * PI * (i + c) / p;
    a * arg.sin() + b
}
```

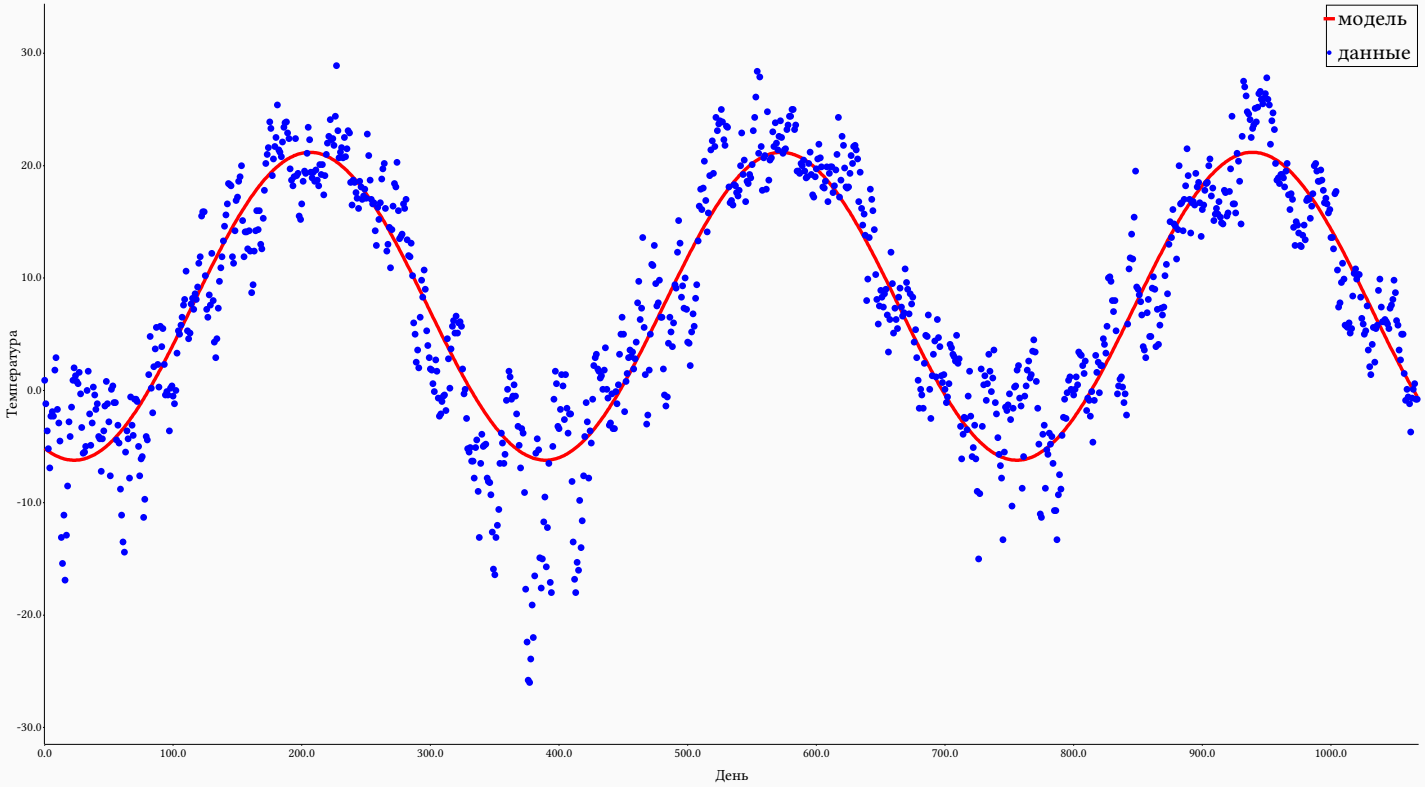
$$L(a, b, p, c) = \sum_{i=0}^{N} (t_i - T_i)^2$$

```
#[autodiff_reverse(d_sse, Const, Active, Active, Active, Active, Active)]
fn sse_loss(temps: &[f32], a: f32, b: f32, p: f32, c: f32) → f32 {
    temps.iter().enumerate().map(|(i, &temp)| {
            let r = model(i as f32, a, b, p, c) - temp;
            r * r
        }).sum()
}
```

Пример обратного прохода

```
// отдаём градиент и loss библиотеке argmin
impl<'a> Gradient for RegressionProblem<'a> {
    fn gradient(&self, param: &Self::Param) → Result<Self::Gradient, Error> {
        /* ... */
        let (loss, da, db, dp, dc) = d_sse(self.temps, a, b, p, c, 1.0);
        /* ... */
    }
}
```

Пример обратного прохода



Питерская норма.

AD артефакты. Пример №1

$$f(x) = x$$
$$f'(x) = 1$$

Правильная реализация:

```
#[autodiff_forward(df, Dual, Dual)]
fn f(x: f32) → f32 {
    x
}

let (v, dx) = df(0.0, 1.0);
// v=0 dx=1
```

AD артефакты. Пример №1

$$f(x) = x$$
$$f'(x) = 1$$

Правильная реализация:

```
#[autodiff_forward(df, Dual, Dual)]
fn f(x: f32) → f32 {
    x
}

let (v, dx) = df(0.0, 1.0);
// v=0 dx=1
```

“Неправильная” реализация:

```
#[autodiff_forward(df, Dual, Dual)]
fn f(x: f32) → f32 {
    if x == 0.0 { 0.0 } else { x }
}

let (v, dx) = df(0.0, 1.0);
// v=0 dx=0
```

AD артефакты. Пример №2

```rust
#[autodiff_forward(d_round, Dual, Dual)]
fn f_round(x: f64) -> f64 {
    x.round()
}

#[autodiff_forward(d_floor, Dual, Dual)]
fn f_floor(x: f64) -> f64 {
    x.floor()
}

#[autodiff_forward(d_ceil, Dual, Dual)]
fn f_ceil(x: f64) -> f64 {
    x.ceil()
}
```

```rust
// x = 1.51
// round(x) = 2,  d/dx round = 0
// floor(x) = 1,  d/dx floor = 0
// ceil(x)  = 2,  d/dx ceil  = 0

// x = 1
// round(x) = 1,  d/dx round = 0
// floor(x) = 1,  d/dx floor = 0
// ceil(x)  = 1,  d/dx ceil  = 0
```

Положение Enzyme в пайплайне rustc

Положение Enzyme в пайплайне rustc. AST

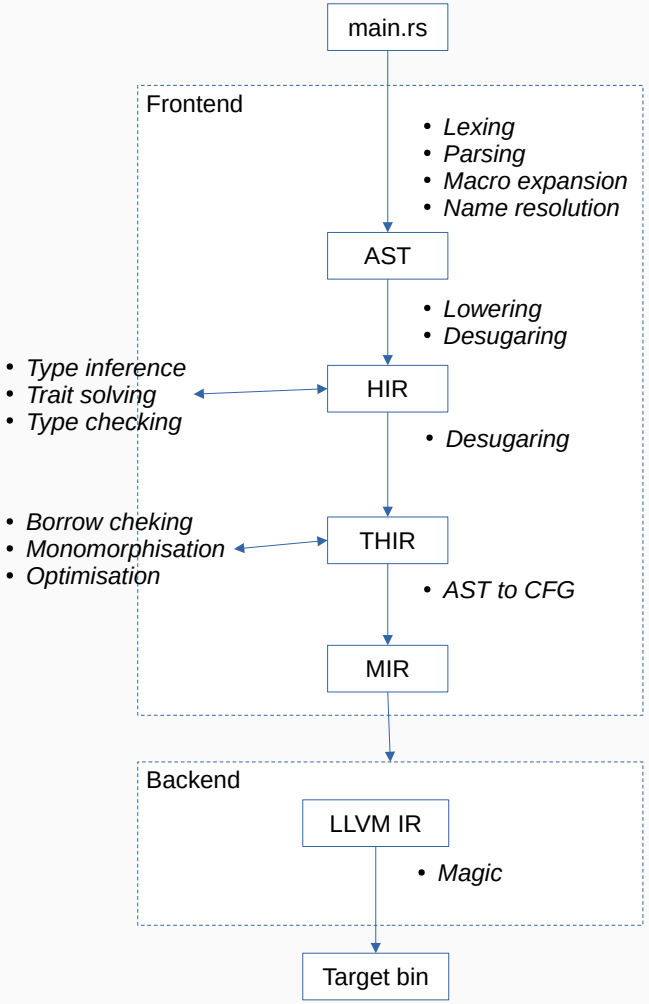
```
#[autodiff_reverse(df, Active, Active)]
fn f(x: f32) → f32 { x * x }
```

↓

```
#[rustc_autodiff] // внутренний атрибут rustc
fn f(x: f32) → f32 { x * x }
```

↓

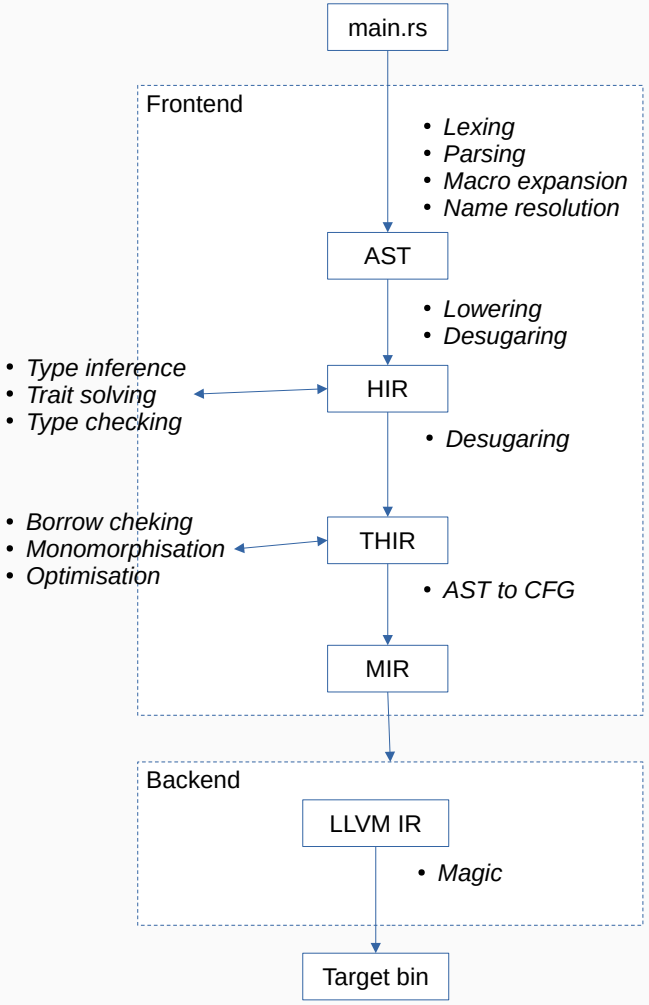
```
fn df(x: f32, df: f32) → (f32, f32) {
    core::intrinsics::autodiff(f, df, (x, df))
}
```



Конвейер rustc. HIR

Атрибуты участвуют в дальнейшем как обычные «компиляторные» атрибуты: их видит HIR.

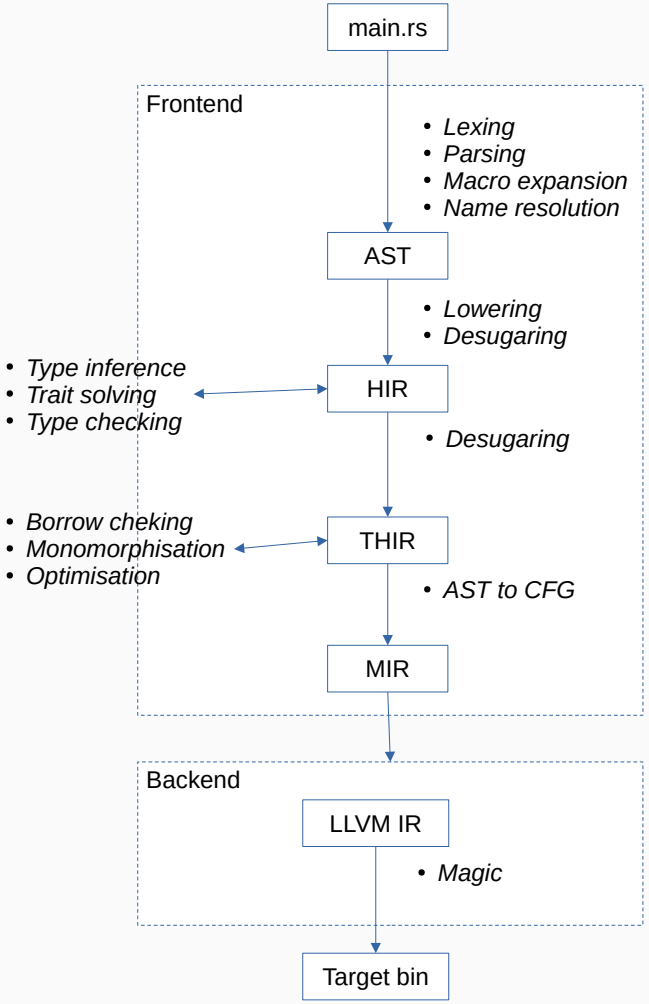
Отдельно ведётся работа по корректному кросс-крейтовому кодированию/декодированию `rustc_autodiff` в метаданных, чтобы `#[autodiff]` работал через зависимости.



Конвейер rustc. THIR, MIR итд

Задача остального пайплайна пронести `rustc_autodiff` дальше к LLVM без изменений

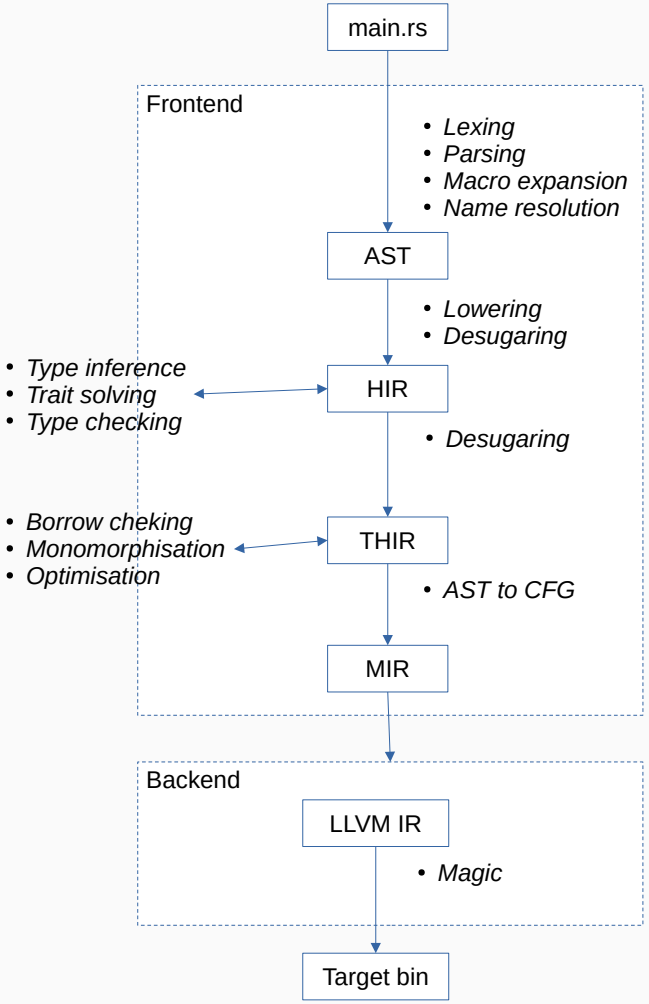
Никаких специальных MIR-пассов под `autodiff` пока нет



Конвейер LLVM

Где-то в середине конвейера оптимизации запускается Enzyme:

- ищет вызовы `__enzyme_autodiff( ... )` (reverse mode) и `__enzyme_fwddiff( ... )` (forward mode)
- на их месте порождает и подставляет градиентные функции
- дальше всё это идёт через обычный LLVM-план оптимизаций



Вывод

Очень перспективно

и

очень сыро

Ограничения Enzyme

Что есть в Enzyme, но пока нет в Rust:

- Кастомные производные для функций
- Batching-режимы
- Кастомные аллокаторы под AD.
- Управление чекпоинтингом/рематериализацией (min-cut и пр.).
- Параллелизм: OpenMP, MPI, и др. Rayon пока нет в Enzyme.
- GPU-дифференцирование из Rust, поддержка CUDA/ROCm.

Ограничения Enzyme:

- Жёсткие требования к «статически анализируемому» коду
- Инструментальность и сообщения об ошибках
- GPU (работы ведутся)

Спасибо за внимание

Оцените доклад

Репа с кодом из доклада

Приложение

Где применяется автодиф?

- Обучение моделей и оптимизация: *градиенты, Гессианы, обратное распространение.*
- Научные расчёты: *чувствительность ODE/PDE, оценка параметров.*
- Управление и робототехника: *траекторная оптимизация, диф. MPC.*
- Графика и зрение: *дифференцируемый рендеринг и инверсная графика.*
- Вероятностное программирование: *вариационный вывод (ADVI).*
- Финансы: *грейки и риск в MC/PDE, калибровка моделей.*
- Инжиниринг и дизайн-оптимизация: *CFD, атмосферные и инженерные задачи.*

Открытые math-LLM

Модель	Веса	Год выпуска
Qwen2.5-Math	1.5B, 7B, 72B	2025
Mathstral	7B	2024
DeepSeekMath	7B	2024
Llemma	7B, 34B	2023
MetaMath	7B, 70B	2023
OpenMath	Mistral-7B, Nemotron-7B	2025
InternLM2-Math-Plus	1.8B, 7B, 20B, 8×22B	2025
OpenThinker2	7B, 32B	2025

Reverse on Forward mode

```
#[autodiff(ddf, Forward, Dual, Dual, Dual, Dual)]
fn df2(x: &[f32;2], dx: &mut [f32;2], out: &mut [f32;1], dout: &mut [f32;1]) {
    df(x, dx, out, dout);
}

#[autodiff(df, Reverse, Duplicated, Duplicated)]
fn f(x: &[f32;2], y: &mut [f32;1]) {
    y[0] = x[0] * x[0] + x[1] * x[0]
}
```

Кастомные функции Julia

Пока отсутствует в Rust

```
function f(y, x)
    y .= x.^2
    return sum(y)
end

# Кастомный forward-rule на Julia
function forward(config::FwdConfig,
                 func::Const{typeof(f)}, # для какой функции
                 ::Type{<:Duplicated},   # аннотация активности результата
                 y::Duplicated, x::Duplicated)
    ret = func.val(y.val, x.val)
    y.dval .= 2 .* x.val .* x.dval
    return Duplicated(ret, sum(y.dval))
end
```

Символьный подход. Оч. сложные

QR:

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R}$$

$$\boldsymbol{Q}^\top \boldsymbol{Q} = \boldsymbol{I}$$

$$\operatorname{diag}(\boldsymbol{R}) > 0$$

$$\boldsymbol{Q}^\top \operatorname{d}(\boldsymbol{Q}) \text{ кососимм.}$$

$$\operatorname{d}(\boldsymbol{A}) = \operatorname{d}(\boldsymbol{Q})\boldsymbol{R} + \boldsymbol{Q}\operatorname{d}(\boldsymbol{R})$$

Cholesky:

$$\boldsymbol{\Sigma} = \boldsymbol{L}\boldsymbol{L}^\top$$

$$\boldsymbol{\Sigma} \succ 0$$

$$\boldsymbol{S} = \left(\boldsymbol{L}^{\{-1\}}\right) \operatorname{d}(\boldsymbol{\Sigma}) \left(\boldsymbol{L}^{\{-1\}}\right)^\top$$

$$\operatorname{d}(\boldsymbol{L}) = \boldsymbol{L}\Phi(\boldsymbol{S})$$

Символьный подход (минусы)

“Взрыв” выражений

$$f_n(x) = \prod_{i=1}^{n} (x + \sin x),$$

$$f_n'(x) = \sum_{k=1}^{n} \left[(1 + \cos x) \cdot \prod_{j=1, j \neq k}^{n} (x + \sin x) \right]$$

при полном разворачивании $\sim n2^{n-1}$ термов

Якобиан, JVP и VJP

$$\mathbb{R}^n \to \mathbb{R}^m$$

$$J_{f(\boldsymbol{x})} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} \in \mathbb{R}^{\{m \times n\}}$$

$\underbrace{J_{f(\boldsymbol{x})} \boldsymbol{e}_k}$

JVP: столбец k при v = e_k

$\underbrace{\boldsymbol{e}_j^{\mathsf{T}} J_{f(\boldsymbol{x})}}$

VJP: строка j при y = e_j

Конечные разности

Градиент по центральным разностям для $f(x_1, x_2) = \sin(x_1) + x_2^2$:

```
use ndarray::array;
use finitediff::ndarr;

let f = |x: &ndarray::Array1<f64>| Ok(x[0].sin() + x[1].powi(2));
let x = array![1.0, 2.0];
let grad = ndarr::central_diff(&f)(&x)?;
```

Якобиан по центральным разностям для $f(\boldsymbol{x}) = \begin{pmatrix} x_1 x_2 \\ x_1 + x_2 \end{pmatrix}$:

```
use numdiff::central_difference as cd;

fn f(x: &[f64]) -> Vec<f64> { vec![x[0] * x[1], x[0] + x[1]] }
let j = cd::jacobian(f, &[1.0, 2.0]);
```

Мат-осведомлённые градиенты Enzyme

```rust
#[no_mangle]
#[inline(never)]
pub unsafe extern "C" fn cblas_dgemm(/* прототип CBLAS из cblas.h */) {
  /* тело, например реализация `dgemm` из крейта faer  */
}
```

`cblas_dgemm` это реализация: $\boldsymbol{C} = \alpha \boldsymbol{A}\boldsymbol{B} + \beta \boldsymbol{C}$

Где смотреть мат-осведомлённые градиенты Enzyme

`main/enzyme/Enzyme/BlasDerivatives.td`

Там декларации такого вида:

```
// C := alpha*op( A )*op( B ) + beta*C
def gemm : CallBlasPattern<(Op $layout, $transa, $transb, $m, $n, $k, $alpha, $A, $lda, $B,
$ldb, $beta, $C, $ldc), ...
```

std::autodiff (reverse mode)

```
#[autodiff_reverse(d_f_affine, Active, Const, Active)]
fn f_affine(x: f64, bias: f64) → f64;
```

↓

```
  pub fn d_f_affine(
      x: f64,
      bias: f64,
      df: f64,
  ) → (f64, f64)

// хотим df/dx, поэтому seed по выходу df = 1.0
let (y, dx) = d_f_affine(x, bias, 1.0);
```

std::autodiff (reverse mode)

```
#[autodiff_reverse(d_f_dot, Duplicated, Duplicated, Active)]
fn f_dot(x: &[f32], w: &[f32]) → f32;
```

```
pub fn d_f_dot(
    x:  &[f32], dx: &mut [f32],
    w:  &[f32], dw: &mut [f32],
) → f32  // возвращаем только значение
```

```
  // буферы для градиентов, должны совпадать по длине с x, w
  let mut dx = [0.0; 3];
  let mut dw = [0.0; 3];
  let y = d_f_dot(&x, &mut dx, &w, &mut dw);
```

std::autodiff (forward mode)

```
#[autodiff_forward(d_f_affine, Dual, Const, Dual)]
fn f_affine(x: f64, bias: f64) → f64;
```

```
fn d_f_affine(
    x: f64, dx: f64,
    bias: f64,
) → (f64, f64)

// хотим df/dx, значит задаём dx = 1.0
let (y, dx) = d_f_affine(x, 1.0, bias);
```

std::autodiff (forward mode)

```
#[autodiff_forward(d_f_dot, Dual, Dual,
Dual)]
fn f_dot(x: &[f32], w: &[f32]) → f32;
```

```
fn d_f_dot(
    x: &[f32], dx: &[f32],
    w: &[f32], dw: &[f32],
) → (f32, f32)
```

```
let dw = vec![0.0; n];
let mut dx = vec![0.0; n];

// Нужно много вызовов чтобы получить
grad_x
for i in 0..n {
    dx[i] = 1.0;
    let (y, df) = d_f_dot(x, &dx, w, &dw);
    grad_x[i] = df;
    dx[i] = 0.0;
}
```

Автодифференцирование

Полное знание — Чёрный ящик

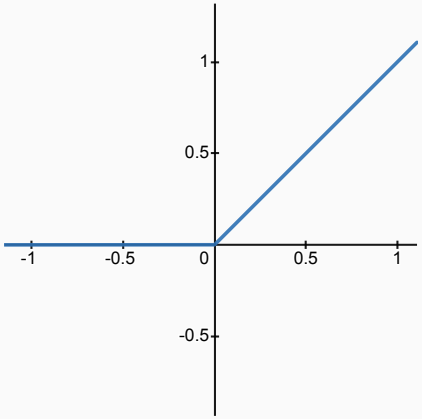
Аналитический подход — Конечная разность

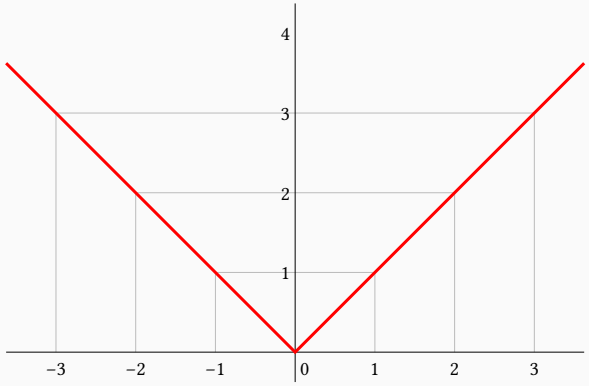
↑

Автодифференцирование

Подход	Точность	Стоимость на $\nabla f(\boldsymbol{x})$
Символьный (аналитический)	Машинная точность (или лучше)	Оценка выведенной формулы
Конечные разности: прямая/обратная	$O(h)$	n вызовов $f()$ (по 1 на координату)
Конечные разности: центральная	$O(h^2)$	$2n$ вызовов $f()$ (по 2 на координату)
Автодифф: прямой режим	Машинная точность	$\sim n \times \text{cost}(f)$ (один проход на вход)
Автодифф: обратный режим	Машинная точность	$\sim 2..4 \times \text{cost}(f)$ (на скалярный выход)

AD артефакты. Пример №3






```rust
#[autodiff_forward(d_relu, Dual, Dual)]
fn relu(x: f32) → f32 {
    if x > 0.0 { x } else { 0.0 }
}

let (v, dx) = d_relu(0.0, 1.0);
// в точке 0 для relu субградиент [0,1],
// а получаем dx=0
```

```rust
#[autodiff_forward(d_abs, Dual, Dual)]
fn abs(x: f32) → f32 {
    if x ⩾ 0.0 { x } else { -x }
}

let (v, dx) = d_abs(0.0, 1.0);
// в точке 0 для abs субградиент [-1,1],
// а получаем dx=1
```