



Embedded Domain Specific Languages

Michelle Ingerl



Gliederung

- Was ist eine eDSL?
- Makros
- Flexible Syntax
- Typinferenz
- Operator Overloading
- Demo
- Fazit
- Quellen



Was ist eine eDSL?

eDSL
Makros
Flexible Syntax
Typinferenz
Operator Overloading
Demo
Fazit
Quellen

Embedded Domain Specific Language

werden in einer Host-Sprache
wie Rust implementiert

spezialisierte Sprachen
für bestimmte Probleme

Sie bieten:

- Natürlichere Syntax
- Typsicherheit durch das Rust-Typsystem
- Kompilierzeit-Überprüfungen

Wofür?

- Verbesserung der Produktivität
- Vereinfachung der Syntax und Semantik
- Ermöglicht Benutzern mit einfachen Befehlen, eigenen Code zu schreiben



Makros

- Ermöglicht die Implementierung von DSL-Syntax
- Compile-time Code-Generierung
- Pattern matching System

```
let input: Expr = math_expr!(5.0 - (2.5 + 3.0));
```

-> Wird in einen Token-Stream zerlegt

```
#[macro_export]  
macro_rules! math_expr { }
```

-> Definiert Muster und Code der bei einem Match ausgeführt wird



```
1  #[macro_export]
2  macro_rules! math_expr {
3      (($inner:tt)*) => {
4          math_expr!($($inner)*)
5      };
6
7      // Base case for numbers
8      ($num:literal) => {
9          crate::math_edsl::Expr::Num($num)
10     };
11
12     // Variable handling
13     (x) => {
14         crate::math_edsl::Expr::Var
15     };
16
17     // Addition
18     ($left:tt + $right:tt) => {
19         crate::math_edsl::Expr::Add(Box::new(math_expr!($left)), Box::new(math_expr!($right)))
20     };
21
22     // Subtraction
23     ($left:tt - $right:tt) => {
24         crate::math_edsl::Expr::Sub(Box::new(math_expr!($left)), Box::new(math_expr!($right)))
25     };
26 }
```

`math_expr!((((((x))))));`

`math_expr!(5.0);`

`math_expr!(2.5 + (1.0 + x));`



Makros

Deklarative Makros (`macro_rules!`, Pattern Matching)

Prozedurale Makros:

- Können Code umschreiben, bevor der Compiler ihn liest
- Werden bei komplexen Code-Generierungen sowie Frameworks benötigt
- `Math_expr!(a + b + c);`

eDSL
Makros
Flexible Syntax
Typinferenz
Operator Overloading
Demo
Fazit
Quellen



Flexible Syntax

eDSL
Makros
Flexible Syntax
Typinferenz
Operator Overloading
Demo
Fazit
Quellen

eDSL ermöglicht uns z.B.:

```
math_expr!((3.0 * (x + 5.0))/(2.0*(sqrt(4.0)))
```

```
math_expr!((4.0 / 2.0) + x);
```

```
math_expr!(4.0 * ((5.0^2.0) - 3.0));
```

```
math_expr!((((((x))))))
```

```
svg_elem!(circle(1, 2, 5, "red"));  
svg_elem!(rect(30, 50, 50, 60, "blue"));  
svg_elem!(line(25, 43, 80, 43, "black"));  
svg_elem!(text(30, 40, 20, "Hello", "yellow"));
```

```
svg_expr!(  
  circle(80, 70, 30, "purple"),  
  rect(30, 50, 50, 60, "blue"),  
  line(25, 43, 80, 43, "black")  
);
```

eDSL ermöglicht uns nicht z.B.:

```
math_expr!(y);  
math_expr!(sin(x));  
math_expr!(x % 2.0);  
math_expr!(2.5 + 3.0 + 1.2);
```

```
svg_elem!(circle(x, 2, 5));  
svg_elem!(circle(80, 70));
```



Typinferenz

Was ist Typinferenz?

Datentypen automatisch bestimmen durch den Rust-Compiler

Vorteile:

- Weniger Code schreiben
- Bessere Lesbarkeit
- Einfacher zu ändern

Grenzen:

- Funktionen benötigen explizite Typangabe
- In komplizierten Fällen ist es manchmal nicht eindeutig

Typinferenz



```
pub enum Expr {  
    Num(f64),
```

```
impl Expr {  
    pub fn eval(&self, x: f64) -> f64 {  
        match self {  
            Expr::Num(n: &f64) => *n,
```

```
($num:literal) => {  
    crate::math_edsl::Expr::Num($num)  
};
```

```
let input: Expr = math_expr!(3.0 + x);
```



Typinferenz

Was ist Typinferenz?

Datentypen automatisch bestimmen durch den Rust-Compiler

Vorteile:

- Weniger Code schreiben
- Bessere Lesbarkeit
- Einfacher zu ändern

Grenzen:

- Funktionen benötigen explizite Typangabe
- In komplizierten Fällen ist es manchmal nicht eindeutig

Operator Overloading



- Ermöglicht Operatoren (wie +, -, *, / etc.) selbst zu definieren
- In Rust durch Traits aus dem `std::ops` Modul

Einschränkungen:

- Keine Erfindung eigener Operatoren möglich (. $+$)
- Feste Reihenfolge bei mathematischen Berechnungen; $a + b * c$
- Symmetrie: für z.B. $a + b$ müssen a und b den gleichen oder einen kompatiblen Datentyp haben
- In Rust durch Traits aus dem `std::ops` Modul implementiert



Operator Overloading

Multiplikation

```
let a = 5 * 3;
```

Dereferenzierung (für Pointer)

```
1 | let x = 5;  
2 | let ptr = &x;  
3 | let y = *ptr;
```

Vektor-Multiplikation

```
1 | use std::ops::Mul;  
2 |  
3 | struct Vektor {  
4 |     x: f64,  
5 |     y: f64,  
6 | }  
7 |  
8 | impl Mul for Vektor {  
9 |     type Output = f64;  
10 |  
11 |     fn mul(self, other: Vektor) -> f64 {  
12 |         self.x * other.x + self.y * other.y  
13 |     }  
14 | }  
15 |  
16 | let v1 = Vektor { x: 2.0, y: 3.0 };  
17 | let v2 = Vektor { x: 4.0, y: 1.0 };  
18 | let ergebnis = v1 * v2;
```

Operator Overloading



- Ermöglicht Operatoren (wie +, -, *, / etc.) selbst zu definieren
- In Rust durch Traits aus dem `std::ops` Modul

Einschränkungen:

- Keine Erfindung eigener Operatoren möglich (`+.+`)
- Feste Reihenfolge bei mathematischen Berechnungen; $a + b * c$
- Symmetrie: für z.B. $a + b$ müssen a und b den gleichen oder einen kompatiblen Datentyp haben
- In Rust durch Traits aus dem `std::ops` Modul implementiert



Demo

Fazit

- Rust bietet durch Makros eine Möglichkeit mit einfachem Syntax eine eDSL zu entwickeln.
- Die gute Typinferenz vereinfacht die Syntax ebenfalls, um das entwickeln einer eDSL zu erleichtern.
- Rust unterstützt Operator Overloading, welches bei z.B. einer mathematischen eDSL hilfreich ist.



Quellen

<https://doc.rust-lang.org/book/>

<https://doc.rust-lang.org/rust-by-example/>

