How Rust Code Implements Kleis Formal Ideas

Kleis Language Project

1 How Rust Code Implements These Formal Ideas

In this section we show how the formal ideas expressed in Kleis (`structure`, `implements`, algebraic data types, pattern matching, parametric indices) can be realized concretely in Rust code.

The goal is not to reproduce Kleis inside Rust, but to demonstrate implementation patterns that closely mirror the formal constructions.

1.1 Structures as Traits

A Kleis structure:

```
structure Monoid(M) {
    operation (•) : M → M → M
    element e : M

    axiom associativity:
      ∀(x y z : M). (x • y) • z = x • (y • z)
}
```

corresponds to a Rust trait:

```
// A Monoid trait in Rust
pub trait Monoid: Sized + Clone {
    fn op(self, other: Self) -> Self;
    fn e() -> Self;
}
```

Here:

- `op` encodes the binary operation $(\cdot)$.
- `e()` encodes the identity element e.
- Trait bounds like `Clone` model practical constraints (copying for tests, etc.) not present in the pure formalism.

The axioms are not enforced by the Rust type system; they are captured as *conventions* plus tests (see below).

1.2 implements as `impl` Blocks

The Kleis declaration:

```
implements Monoid(ℤ) {
    operation (•) = builtin_add
    element e = 0
}
```

is naturally implemented as:

```
impl Monoid for i64 {
    fn op(self, other: Self) -> Self {
        self + other
    }

    fn e() -> Self {
        0
    }
}
```

This is a direct correspondence:

- `implements Monoid(T)` ↔ `impl Monoid for T`.
- `operation` assignments ↔ method bodies.

1.3 Extending Structures via Trait Inheritance

A Kleis extension:

```
structure Group(G) extends Monoid(G) {
    operation inv : G → G
}
```

maps to a Rust trait that *extends* another trait:

```
// Group extends Monoid
pub trait Group: Monoid {
    fn inv(self) -> Self;
}
```

The relationship:

`extends Monoid(G)` ⟷ `trait Group: Monoid { ... }`

In this way:

- Every `Group` is also a `Monoid`.
- The Rust compiler enforces that any `impl Group for T` must also satisfy `Monoid`'s method requirements.

1.4 Algebraic Data Types as Enums

A Kleis algebraic data type:

```
data Option(T) = None | Some(T)
```

is directly expressible as a Rust enum:

```
pub enum Option<T> {
    None,
    Some(T),
}
```

A more complex example:

```
data Result(E, T) = Err(E) | Ok(T)
```

becomes:

```
pub enum Result<E, T> {
    Err(E),
    Ok(T),
}
```

Thus:

- Kleis ADTs $\leftrightarrow$ Rust `enum`s.
- Constructors $\leftrightarrow$ enum variants.
- Type parameters preserved as generics.

1.5 Pattern Matching as `match` Expressions

Kleis:

```
match x {
    None    => 0
  | Some(v) => v
}
```

Rust:

```
match x {
    Option::None      => 0,
    Option::Some(v)   => v,
}
```

Or with the standard prelude:

```
match x {
    None      => 0,
    Some(v)   => v,
}
```

Key points:

- Rust `match` enforces *exhaustiveness*, as in Kleis.
- Nested constructor patterns are supported in the same way.
- Non-exhaustive matches are compile-time errors.

1.6 Parametric Types and Indexing via Generics and Const Generics

Kleis may define a matrix type:

```
data Matrix(m: Nat, n: Nat, T)
```

Rust can approximate this with const generics:

```
// A dimension-indexed matrix in Rust
pub struct Matrix<T, const M: usize, const N: usize> {
    data: [[T; N]; M],
}
```

Matrix multiplication in Kleis:

```
operation multiply :
  Matrix(m, n, T) → Matrix(n, p, T) → Matrix(m, p, T)
```

can be encoded in Rust as:

```
impl<T, const M: usize, const N: usize, const P: usize>
    Matrix<T, M, N>
where
    T: Copy + std::ops::Add<Output = T> + std::ops::Mul<Output = T> + Default,
{
    pub fn multiply(self, rhs: Matrix<T, N, P>) -> Matrix<T, M, P> {
        let mut result = Matrix::<T, M, P>::zero();
        for i in 0..M {
            for j in 0..P {
                let mut sum = T::default();
                for k in 0..N {
                    sum = sum + self.data[i][k] * rhs.data[k][j];
                }
```

```
                result.data[i][j] = sum;
            }
        }
        result
    }

    pub fn zero() -> Self {
        Matrix {
            data: [[T::default(); N]; M],
        }
    }
}
```

This respects the same dimension constraints:

$$(m \times n) \cdot (n \times p) \to (m \times p).$$

1.7 Vector Spaces and `over Field(F)` via Trait Bounds

Kleis:

```
structure Field(F) { ... }

structure VectorSpace(V) over Field(F) {
    operation (+) : V → V → V
    operation (·) : F → V → V
}
```

Rust:

```
pub trait Field:
    Sized + Copy
    + std::ops::Add<Output = Self>
    + std::ops::Sub<Output = Self>
    + std::ops::Mul<Output = Self>
    + std::ops::Div<Output = Self>
{
    fn zero() -> Self;
    fn one() -> Self;
}

pub trait VectorSpace<F: Field>: Sized + Copy {
    fn add(self, other: Self) -> Self;
    fn smul(scalar: F, v: Self) -> Self;
}
```

Here:

- The `Field` trait encodes the base structure.
- `VectorSpace<F>` is parametrized by a type `F` that must satisfy `Field`.
- This is a direct analogue of "`over Field(F)`" in Kleis.

1.8 Nested Structures via Traits with Associated Types

Kleis:

```
structure Ring(R) {
    structure additive : AbelianGroup(R) { ... }
    structure multiplicative : Monoid(R) { ... }
}
```

Rust approximation using associated traits:

```
// Additive group structure
pub trait AdditiveGroup {
    type Carrier;
    fn add(x: Self::Carrier, y: Self::Carrier) -> Self::Carrier;
    fn zero() -> Self::Carrier;
    fn neg(x: Self::Carrier) -> Self::Carrier;
}

// Multiplicative monoid structure
pub trait MultiplicativeMonoid {
    type Carrier;
    fn mul(x: Self::Carrier, y: Self::Carrier) -> Self::Carrier;
    fn one() -> Self::Carrier;
}

// Ring ties them together
pub trait Ring:
    AdditiveGroup<Carrier = <Self as Ring>::Carrier>
    + MultiplicativeMonoid<Carrier = <Self as Ring>::Carrier>
{
    type Carrier;
}
```

Or more compactly:

```
pub trait Ring {
    type Carrier;
```

```
    fn add(x: Self::Carrier, y: Self::Carrier) -> Self::Carrier;
    fn zero() -> Self::Carrier;
    fn neg(x: Self::Carrier) -> Self::Carrier;

    fn mul(x: Self::Carrier, y: Self::Carrier) -> Self::Carrier;
    fn one() -> Self::Carrier;
}
```

This encodes the same idea: a ring structure consists of two compatible substructures on a single carrier set.

1.9 Laws (Axioms) as Test Suites

Kleis can write axioms:

```
axiom associativity:
  ∀(x y z : M). (x • y) • z = x • (y • z)
```

Rust cannot enforce this axiom in the type system, but can encode it as property tests, for example using `proptest`:

```
#[cfg(test)]
mod tests {
    use super::*;
    use proptest::prelude::*;

    proptest! {
        #[test]
        fn monoid_associativity(x in any::<i64>(),
                                y in any::<i64>(),
                                z in any::<i64>()) {
            let lhs = Monoid::op(Monoid::op(x, y), z);
            let rhs = Monoid::op(x, Monoid::op(y, z));
            prop_assert_eq!(lhs, rhs);
        }
    }
}
```

Thus:

- The Kleis axiom becomes a PBT property.
- The implementation is "lawful" if all such tests pass.

1.10 Summary

Rust provides:

- `trait` ≈ Kleis `structure`,
- `impl` ≈ Kleis `implements`,
- `enum` ≈ Kleis `data` (ADT),
- `match` ≈ Kleis `match`,
- generics + const generics ≈ Kleis type parameters and indices,
- trait bounds ≈ Kleis constraints / `over` clauses.

The main difference is that Kleis can *talk about* axioms and algebraic properties as first-class citizens in the language, whereas Rust implementations encode these properties through carefully structured traits and test suites.

In practice, Rust code can serve as a *back-end* implementation of formal Kleis specifications: Kleis describes the abstract algebraic object, and Rust provides the concrete, testable realization of that object as executable code.