From Theory to Practice

Bridging Kleis Semantics and Rust Implementations

Kleis Language Project

1 From Theory to Practice: Bridging Kleis Semantics and Rust Implementations

Kleis is defined as a mathematically precise language whose semantics are given by inference rules, algebraic structures, and type-theoretic principles. However, practical implementations—for example, in Rust—must realize these abstract notions as executable code. This section explains how each theoretical notion in Kleis corresponds to a concrete programming construct, showing how abstract semantics guides low-level implementation decisions.

1.1 1. Structures as Implementable Interfaces

In the formalism, a `structure` is a tuple of operations and elements together with a collection of axioms:

$$\frac{\forall x, y, z \in M.\ (x \cdot y) \cdot z = x \cdot (y \cdot z)}{\texttt{structure Monoid(M)}\ \{ \cdots \}}$$

The semantics treats structures as *records of functions* with behavioral laws.

In practice, Rust realizes this as:

- a *trait* specifying the operations,
- one or more *impl blocks* providing concrete functions,
- unit and property tests approximating the axioms.

Thus, structures move from "semantic entities with axioms" to "trait interfaces with verifiable behavior."

1.2 2. Implementations as Semantic Instantiations

The formal semantics interprets:

$$\texttt{implements Monoid}(T)$$

as: "Provide the interpretation of each operation of the `Monoid` structure for the type T, ensuring the axioms hold."

Rust instantiates this via:

```
impl Monoid for i64 { ... }
```

The compiler verifies:

- that all required operations are implemented,
- that the signatures match.

The axioms—associativity, identity—cannot be encoded in Rust's type system; thus, they are validated through:

- symbolic property tests,
- documentation of lawfulness,
- static analysis (e.g. dimension checks for matrices).

Hence, implementation corresponds to *semantic instantiation*, and testing corresponds to *axiom validation*.

1.3 3. Algebraic Data Types and Constructors

The formal Kleis syntax:

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

is modeled in the semantics as a disjoint sum type:

$$\mathsf{Option}(T) \cong 1 + T.$$

Rust realizes this as:

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

Constructor semantics is preserved exactly:

- `None` is a nullary constructor (1),
- `Some` is a unary constructor.

The operational semantics rule:

$$\frac{}{\rho \vdash C(v_1, \ldots, v_k) \Downarrow C(v_1, \ldots, v_k)}$$

maps directly to Rust's cost-free enum construction.

Thus, data constructors in Kleis correspond to zero-overhead tagged unions in Rust.

1.4 4. Pattern Matching and Exhaustiveness

Kleis pattern matching is defined by a judgment:

$$\mathsf{match}(p, v) = \theta$$

and an operational rule:

$$\frac{\rho \vdash e \Downarrow v \qquad \mathsf{firstMatch}(v, p_i \Rightarrow e_i) = (\theta, e_k)}{\rho \vdash \texttt{match}\ e\{\cdots\} \Downarrow v'}$$

Rust implements this directly:

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

Rust enforces:

1. *exhaustiveness*, corresponding to the Kleis judgment $\mathsf{Exh}(T, \{p_1, \ldots, p_n\})$,
2. *non-redundancy*, corresponding to the Kleis $\mathsf{NR}(T, \vec{p})$ analysis.

In Kleis, these are semantic judgments and algorithmic procedures. In Rust, they are enforced by the compiler.

Thus, the semantics of pattern matching transitions almost perfectly into Rust's `match` system.

1.5 5. Parametric Types and Indices

Index-polymorphic types such as:

$$\texttt{Matrix}(m, n, T)$$

formally behave as families indexed by natural numbers:

$$\mathsf{Matrix} : \mathbb{N} \times \mathbb{N} \times \mathsf{Type} \to \mathsf{Type}.$$

The Kleis type checker ensures dimension-correct operations:

$$\texttt{Matrix}(m, n, T) \times \texttt{Matrix}(n, p, T) \to \texttt{Matrix}(m, p, T).$$

Rust implements this via *const generics*:

```
struct Matrix<T, const M: usize, const N: usize> {
    data: [[T; N]; M],
}
```

The Rust compiler enforces dimension correctness:

```
fn multiply<const M: usize, const N: usize, const P: usize>(
    a: Matrix<T, M, N>,
    b: Matrix<T, N, P>)
-> Matrix<T, M, P> { ... }
```

Thus, type-level indices become const generics in Rust.

1.6 6. Laws and Axioms in Practice

Kleis includes formal axioms:

$$\forall x, y, z.\ (x \cdot y) \cdot z = x \cdot (y \cdot z).$$

The semantics treats axioms as constraints on valid implementations.
Rust cannot encode axioms in its type system, but it bridges the gap by:

- property tests (`proptest`),
- fuzzing strategies,
- runtime assertions,
- documentation stating the "lawful" contract.

Thus, axioms move from *logical requirements* to *programmable verification artifacts*.

1.7 7. Operational Semantics to Executable Code

A big-step rule such as:

$$\frac{\rho \vdash e_1 \Downarrow \lambda x.e \qquad \rho \vdash e_2 \Downarrow v \qquad \rho[x \mapsto v] \vdash e \Downarrow v'}{\rho \vdash e_1\, e_2 \Downarrow v'}$$

corresponds to Rust's function-call mechanism:

```
let f = |x| x + 1;
let result = f(41);
```

But unlike Kleis:

- Rust is call-by-value only,
- Kleis semantics is defined independently of evaluation strategy,
- Mathematically, Kleis supports symbolic evaluation through axioms.

Still, the mapping is clean: semantic application $\Downarrow$ concretizes as a Rust function call.

1.8 8. Summary: A Bidirectional Bridge

We summarize the connections:

Kleis Formal Concept	**Rust Implementation**
Structure	Trait
Implements	Impl block
ADT constructor	Enum variant
Pattern match	`match` expression
Indices (m, n)	Const generics
Type abstraction	Type parameters
Axioms	Property tests
Operational semantics	Rust runtime behavior
Exhaustiveness	Rust compiler checks
Non-redundancy	Rust warning system

Thus, the theoretical constructs of Kleis admit a clear and faithful implementation in Rust, preserving the mathematical structure while remaining fully executable and efficient.