A Category-Theoretic Guide to Kleis

Kleis Language Project

Contents

Preface

Kleis is a language whose structural constructs correspond directly to category-theoretic notions. It does not merely imitate the syntax of mathematics; rather, it encodes mathematical structure in a form suitable for computation, formal reasoning, and algebraic abstraction.

This document introduces Kleis from the point of view of category theory:

- structures as algebraic theories,
- implementations as models,
- `extends` as morphisms of theories,
- `over` as fibrations or indexed categories,
- pattern matching as initiality of inductive types,
- type constructors as functors,
- polymorphism as naturality,
- axioms as commutative diagrams.

We assume familiarity with basic category theory, universal algebra, and type theory.

1 Structures as Algebraic Theories

In Kleis, a **structure** represents a finitary algebraic theory in the sense of Lawvere.

Example:

```
structure Monoid(M) {
    operation (•) : M × M → M
    element e : M
    axiom left_identity:
        ∀(x : M). e • x = x
}
```

This corresponds to a Lawvere theory $\mathsf{Th}(\mathsf{Monoid})$ with:

- one basic sort M,
- one binary operation symbol $\mu : M \times M \to M$,
- one constant $e : 1 \to M$,
- equations expressing associativity and identity.

Formally:

$$\mathsf{Th}(\mathsf{Monoid}) : \mathbf{FinSet} \to \mathbf{Set},$$

a finitary product-preserving functor.

2 `implements` as Models of Theories

Kleis uses:

```
implements Monoid(ℕ) { ... }
```

to express that the set $\mathbb{N}$ with the given operations forms a model of $\mathsf{Th}(\mathsf{Monoid})$.

That is, an implementation is a functor:

$$\mathsf{Mod}(\mathsf{Monoid}) = \mathbf{Set}^{\mathsf{Th}(\mathsf{Monoid})}.$$

Every implementation is a product-preserving functor sending the abstract operations to actual functions.

Thus:

- `structure` = algebraic theory T,
- `implements` = interpretation $F : \mathsf{T} \to \mathbf{Set}$.

3 `extends` as Morphisms of Theories

A declaration such as:

```
structure Group(G) extends Monoid(G) { ... }
```

is a *morphism of Lawvere theories.*
There is an inclusion:

$$\mathsf{Th}(\mathsf{Monoid}) \hookrightarrow \mathsf{Th}(\mathsf{Group}).$$

Thus every group model restricts to a monoid model:

$$\mathsf{Mod}(\mathsf{Group}) \to \mathsf{Mod}(\mathsf{Monoid}).$$

This is a forgetful functor:

$$U : \mathbf{Grp} \to \mathbf{Monoid}.$$

In Kleis:

- `extends` imports operations and axioms,
- the induced inclusion corresponds to a Cartesian morphism.

4 `over` as Indexed Structures and Fibrations

Kleis allows:

```
structure VectorSpace(V) over Field(F) { ... }
```

This expresses the classical fact that vector spaces form a *category fibred over the category of fields*:

$$\pi : \mathbf{Vect} \to \mathbf{Field}.$$

Given a field F, the fiber $\pi^{-1}(F)$ is the category of vector spaces over F.
Thus `over` introduces a fibration of theories:

$$\mathsf{Th}(\mathsf{VectorSpace}) \to \mathsf{Th}(\mathsf{Field}).$$

This is an indexed family of theories parametrized by the base theory.

5 `where` as Logical Predicates and Subfibrations

A future Kleis feature:

```
implements Ring(R) where Commutative(R)
```

corresponds to restricting to models satisfying a predicate.
Categorically, this forms a *subfibration* of the original fibration:

$$\mathbf{CRing} \hookrightarrow \mathbf{Ring}.$$

Such constraints are represented as:

$$\mathsf{Mod}(\mathsf{Ring}) \supseteq \mathsf{Mod}(\mathsf{Ring})_\varphi$$

where φ is the predicate (e.g. commutativity).

6 Nested Structures as Internal Subtheories

Consider:

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

This corresponds to an amalgamated sum of theories:

$$\mathsf{Th}(\mathsf{Additive}) \sqcup \mathsf{Th}(\mathsf{Multiplicative}) \longrightarrow \mathsf{Th}(\mathsf{Ring}).$$

Categorically, a ring is an object with two compatible algebra structures. This corresponds to an internal diagram of theories.

7 Algebraic Data Types as Initial Algebras

Consider:

```
data List(T) = Nil | Cons(T, List(T))
```

This defines the initial algebra of a functor:

$$F(X) = 1 + T \times X.$$

Pattern matching corresponds to the universal property:

$$\text{Every } F\text{-algebra receives a unique morphism from } \mu F.$$

That is, `match` is a catamorphism.

8 Type Constructors as Functors

A declaration such as:

```
Matrix(m,n,T)
```

behaves as a functor:

$$\mathsf{Matrix}_{m,n} : \mathbf{Type} \to \mathbf{Type}.$$

Polymorphic functions:

```
∀(T). f : T → T
```

are *natural transformations*:

$$f : \mathrm{Id} \Rightarrow \mathrm{Id}.$$

More generally:

$$f : F \Rightarrow G$$

for functors F, G corresponding to type expressions.

9 Pattern Matching as Case Analysis from an Initial Object

Given a data type:

$$D = \sum_i C_i(\vec{A}_i),$$

a match-expression corresponds to a morphism:

$$D \to X$$

obtained by specifying morphisms for each constructor branch.

This is the same universal property as in inductive type theory and initial algebras in **Set**.

10 Axioms as Commutative Diagrams

A Kleis axiom:

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

corresponds to requiring the following diagram commute:

$$\begin{array}{ccc} (M \times M) \times M & \xrightarrow{\mu \times \mathrm{id}} & M \times M \\ \downarrow_{\alpha} & & \downarrow_{\mu} \\ M \times (M \times M) & \xrightarrow{\mathrm{id} \times \mu} & M \end{array}$$

Patterns and axioms in Kleis are thus expressed purely diagrammatically.

11 The Category of Kleis Structures

Kleis defines a category:

$$\mathbf{KleisTh}$$

whose objects are theories (structures), and whose morphisms are `extends`-maps (theory inclusions).

Models

For each theory T, there is a category of models:

$$\mathsf{Mod}(T) = \mathbf{Set}^T.$$

The entire semantics of Kleis can be viewed as a fibration:

$$\mathsf{Mod} : \mathbf{KleisTh}^{op} \to \mathbf{Cat}.$$

12 Functorial Semantics of Kleis Programs

A program defines:

- a theory T (from `structure` declarations), and
- a model M (from `implements` declarations).

Evaluating a program corresponds to:

$$\text{Computing } M \in \mathsf{Mod}(T).$$

Thus running a Kleis program is applying a functor that interprets the syntactic theory as a semantic object in **Set**.

13 Kleis as an Internal Language of a Fibration

The constructs `extends`, `over`, `implements`, and `where` correspond precisely to:

- morphisms of theories,
- fibrational indexing,
- sections/choices of models,
- subfibrations determined by predicates.

This positions Kleis alongside dependently typed languages and algebraic specification systems such as:

$$\text{Lawvere theories}, \quad \text{Sketches}, \quad \text{Institution theory}.$$

14 Conclusion

Kleis is not merely a typed functional language. It is a categorical metalanguage for describing algebraic theories, their morphisms, and their models.

In summary:

- **structure** = algebraic theory,
- **extends** = morphism of theories,
- **over** = fibration or indexed theory,
- **nested structures** = internal diagrams,
- **data types** = initial algebras,
- **polymorphism** = naturality,
- **implements** = model in **Set**,
- **evaluation** = interpretation functor.

Kleis thus forms a bridge between category theory, type theory, and executable algebraic computation.