Kleis Language Specification

Version 0.6 (Updated December 2024)

Kleis Project

Contents

1 Lexical Structure **3**

2 Programs and Declarations **3**

3 Algebraic Data Types **3**

4 Pattern Matching **4**

5 Types **4**

6 Structures **5**

7 `extends`: Inheritance of Structure **5**

8 `over`: Parameterized Structures **5**

9 `implements`: Concrete Models **6**

10 `where`: Logical Conditions **6**

11 Nested Structures **6**

12 Functions and Definitions **6**

13 Expressions **7**

14 Typing Rules (Informal) **7**

15 Operational Semantics (Sketch) **8**

16 Examples **8**

17 Conclusion **9**

Preface

Kleis is a structurally rich, algebra-oriented language designed to express mathematical objects, algebraic theories, axioms, and executable definitions within a unified formal system.

The guiding principles of Kleis are:

- **Mathematical correctness**: structures correspond directly to algebraic theories.
- **Self-hosting**: the language can describe its own type system.
- **Parametric generality**: all operators and structures are parametric in types and indices.
- **Strong static typing**: every construct is typed, including axioms and propositions.
- **No implicit behavior**: all operations must be explicitly defined or implemented.

This document presents the syntax, typing rules, semantic intuition, and structural foundations of Kleis.

1 Lexical Structure

Identifiers:

$$\texttt{identifier} ::= \texttt{letter}(\texttt{letter} \mid \texttt{digit} \mid \_)^*$$

Greek lowercase letters denote type variables:

$$\alpha,\ \beta,\ \gamma,\ \ldots$$

Numbers:

$$\texttt{number} ::= \texttt{integer} \mid \texttt{decimal} \mid \texttt{scientific}$$

Strings use double quotes:

"hello".

Comments:

- `// line comment`
- `/* block comment */`

Whitespace is ignored except where required for separation.

2 Programs and Declarations

A Kleis program consists of a sequence of declarations:

```
program ::= { declaration } ;
```

Declarations include data types, structures, function definitions, implementations, and type aliases.

3 Algebraic Data Types

Algebraic data types follow the form:

```
data Option(T) = None | Some(T)
data Bool = True | False
data Pair(A,B) = Pair(A,B)
```

Formally:

$$\texttt{data}\ \ D(\vec{T}) = C_1(\vec{A_1}) \mid C_2(\vec{A_2}) \mid \cdots$$

Constructors introduce sum types.

4 Pattern Matching

Pattern matching provides decomposition of algebraic values.

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

Patterns include:

- wildcard: `_`
- variable: `x`
- constructor pattern: `Some(x)`
- constant: `0`, `"hello"`, `True`

Exhaustiveness checking is performed statically.

5 Types

Primitive types:

$$\mathbb{R},\ \mathbb{C},\ \mathbb{Z},\ \mathbb{N},\ \texttt{Bool},\ \texttt{String}.$$

Function types:

$$A \to B.$$

Parametric types:

$$T(X_1, \ldots, X_n).$$

Polymorphic types use universal quantifiers:

```
(T : Type). T → T
```

Constraints may appear:

```
(T). Semigroup(T)  T × T → T
```

6 Structures

A structure defines an algebraic theory:

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

A structure consists of:

- operations,
- elements,
- axioms,
- optionally nested structures,
- optionally a parameter clause.

7 `extends`: Inheritance of Structure

Kleis models mathematical extension of theories:

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

Meaning:

$$\text{Group axioms } \supset \text{Monoid axioms.}$$

All members of the parent structure are imported.

8 `over`: Parameterized Structures

Many structures depend on a base structure, e.g. vector spaces over fields:

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

Mathematically:

$$(V, +, \cdot) \text{ is a vector space over } F.$$

9 `implements`: Concrete Models

To assert that a concrete type satisfies a structure:

```
implements Field() {
    element zero = 0
    element one = 1
    operation (+) = builtin_add
}
```

This is analogous to writing:

$$\mathbb{R} \text{ is a field.}$$

10 `where`: Logical Conditions

Implementations or operations may require side-conditions:

```
operation det : Matrix(n,n) →
```

implicitly requires $n = n$ (square matrices).
Future versions allow:

```
implements Ring(T) where Commutative(T)
```

11 Nested Structures

A structure may contain substructures:

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

This mirrors standard algebra:

$$(R, +, 0) \text{ is an Abelian group,} \quad (R, \cdot, 1) \text{ is a monoid.}$$

12 Functions and Definitions

Functions may be defined with or without parameters:

```
define id(x : T) = x
define square(x) :  = x * x
```

Definitions inside structures extend the algebraic signature (Grammar v0.6):

```
structure Ring(R) {
    operation (+) : R × R → R
    operation negate : R → R
    operation (×) : R × R → R

    // Derived operation with default implementation
    operation (-) : R × R → R
    define (-)(x, y) = x + negate(y)
}
```

This allows structures to provide default implementations of derived operations.

13 Expressions

Expressions include:

- identifiers: x
- literals: numbers, strings, booleans
- function application: $f(x)$
- infix operators: $x + y$, AB
- prefix/postfix operators: $-x$, A^T
- lambda expressions: $\lambda x.\, e$
- let-bindings
- conditionals
- match-expressions

14 Typing Rules (Informal)

Variables

$$\Gamma(x) = T \implies \Gamma \vdash x : T$$

Application

$$\Gamma \vdash f : A \to B, \quad \Gamma \vdash x : A \quad \Rightarrow \quad \Gamma \vdash f(x) : B$$

Pattern Matching

Constructor patterns refine the typing context:

$$\texttt{Some}(x) : \texttt{Option}(T) \quad \Rightarrow \quad x : T.$$

15 Operational Semantics (Sketch)

Kleis is not defined by a reduction calculus, but by:

- evaluation of expressions,
- dispatch to implementations for operations,
- symbolic handling of axioms and propositions,
- static enforcement of typing rules.

Evaluation is strict and call-by-value.

16 Examples

Matrix Multiplication

```
structure MatrixMultipliable(m,n,p,T) {
    operation multiply :
        Matrix(m,n,T) → Matrix(n,p,T) → Matrix(m,p,T)
}

implements MatrixMultipliable(m,n,p,) {
    operation multiply = builtin_matrix_multiply
}
```

Vector Space

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

implements VectorSpace(Vector(n)) over Field() {
    operation (+) = vector_add
    operation (·) = scalar_vector_mul
}
```

17 Conclusion

Kleis unifies algebraic specification, type theory, and executable semantics into a single mathematical-programming language. Its structural foundations map closely to mathematical practice: extensions of theories, parameterized structures, concrete models, and logical constraints.

This specification is intended as a basis for both implementation and formal reasoning.