Bliss Language Specification
Data and Action Model

Mayukh Chakraborty

Abstract

Bliss is a programming language whose design is driven by a single constraint: meaning must be chosen explicitly and must never be inferred by the compiler. This paper specifies the core model that enforces that constraint, separating *representation* (raw data), *behavior* (actions), and *meaning* (binded semantic types). We define open versus closed semantics, value-level composition of actions, interpretation tokens created by `bind`, and the calling conventions that determine which semantic layer survives a function boundary. The goal is a mechanically checkable system that prohibits implicit widening, global coherence, and accidental semantic drift, while still enabling explicit reinterpretation when desired.

1 Design Principles

Bliss is built around one central idea:

Meaning is chosen, never inferred.

From this follow several non-negotiable principles:

- No implicit behavior
- No implicit meaning
- No semantic widening
- No global coherence rules
- All semantic changes are explicit and local

The type system exists to enforce these principles mechanically.

2 Type System Overview

Bliss has **three distinct kinds of types**, all first-class and mutually disjoint:

1. **Raw Data Types** — memory only

2. **Action Types** — behavior only

3. **Binded (Semantic) Types** — memory + behavior + meaning

None of these is derived implicitly from another. Transitions between them must be explicit.

3 Raw Data Types

3.1 Definition

A *data type* defines memory layout only.

```
data Data {
    hide u64 byte_ptr;
    private u32 size;
}
```

Properties:

- Define storage and representation
- Contain no behavior
- Carry no meaning
- Expose no methods
- Possess **open semantics**

A value of a data type is called *raw data*:

```
Data a;
```

3.2 Raw Data Semantics

Raw data values expose no members:

```
a.length(); // error
```

However, raw data may be *reinterpreted explicitly*:

```
a::X.length(); // valid
a::Y.length(); // valid
```

Rule. Raw data has no inherent behavior, but may be interpreted under any action or bind.

4 Action Types

4.1 Definition

An *action* represents a behavioral capability, independent of storage and meaning.

```
action X {
    fx length(self) -> int;
}
```

Properties:

- Have no storage
- Have no instances
- Are not data
- Do not define meaning
- May be used as parameter types

Actions describe *required behavior*, not semantic intent.

4.2 Action Annotations

Actions may be annotated to refine validity rules:

```
@reinterpret
action UTF8 {
    fx length(self) -> int;
}

@composable
action Printable {
    fx print(self);
}
```

Annotations:

- Affect validity and usage rules
- Never affect memory representation

5 Binded (Semantic) Types

5.1 Definition

A *bind* specifies how an action (or a set of actions) is interpreted for some data. The binding produces an *interpretation token* that can be used to attach meaning to raw data.

```
bind X, Data as SomeType {
    fx length(self) -> int { ... }
}
```

Facts:

- It is neither `Data` nor `X`
- It participates fully in the type system
- It has **closed semantics**

5.2 Construction of Binded Values

A binded value is constructed explicitly:

```
Data::SomeBinding b;
Data::(BindA, BindB) c; // multi bindings
```

This is *semantic construction*, not reinterpretation.

6 Open vs. Closed Semantics

6.1 Open Semantics (Raw Data)

```
Data a;
```

Characteristics:

- No meaning
- No guarantees
- Reinterpretation allowed

```
a::X.length();
a::Y.length();
```

6.2 Closed Semantics (Binded Types)

```
Data::SomeBind b;
```

Characteristics:

- Meaning is fixed
- Semantics are sealed
- Reinterpretation is forbidden

```
b.length(); // valid
b::X.length(); // error
```

Rule. Only raw data may be reinterpreted.

7 Value-Level Action Composition

A value may expose multiple actions simultaneously:

```
Type::(X, Y) k;
```

This means:

- Underlying data is unchanged
- Both actions are available
- No new semantic type is created
- Composition is local to the value

Rules:

- Composition does not create a bind
- No additional actions may be introduced later
- Ambiguity must be resolved explicitly

```
k.length::X();
k.length::Y();
```

8 Binding and Interpretation Tokens

A *bind* associates a non-empty set of actions with a data type by creating an *interpretation token*. An interpretation token does not introduce a new nominal data type; instead it establishes a sealed semantic relationship between:

- a specific data type (memory), and
- a specific unordered set of actions (behavior).

```
bind (X, Y, Z), Data as A;
```

Here, A is an interpretation token representing:

```
InterpretationToken = (Data, ActionSet)
```

Actions themselves are not interpretation tokens:

- `X` is an action.
- `Data` is memory.
- `A` is the interpretation token joining them.

Therefore:

```
Data::A // valid
Data::X // illegal
Data::(X,Y) // illegal
```

Only interpretation tokens may be used with the `::` operator.

8.1 Binding Rules

1. The action set must be non-empty.
2. The action set is unordered.
3. Duplicate actions are illegal.
4. The action set is sealed at bind time.
5. No additional actions may be introduced after binding.
6. An interpretation token is uniquely identified by its data type and exact action set.
7. Binding does not imply subtyping relationships.

8.2 Identity of Interpretation Tokens

Interpretation tokens are *nominal declarations*. Each `bind` statement introduces a distinct interpretation token, even if the underlying data type and action sets are identical:

```
bind (X), Data as A;
bind (X), Data as B;
```

`A` and `B` are distinct interpretation tokens. They represent separate semantic commitments, even though they relate the same data type and action set. Identity is therefore defined by *declaration identity*, not structural equivalence.

8.3 Closed Semantics

An interpretation token defines a closed semantic universe. For a value:

```
Data::A v;
```

- Only actions in the token's action set are accessible.

- Reinterpretation into additional actions is forbidden.
- Semantic widening is illegal.

8.4 Prohibited Forms

The following are illegal:

```
bind (), Data as A; // empty action set
bind (X, X), Data as A; // duplicate actions
Data::X; // action is not a token
Data::(X,Y); // actions are not tokens
```

Binding never implies structural subtyping, automatic widening, or action inference.

9 Method Resolution

9.1 Unqualified Calls

```
value.method();
```

Legal only if:

- the value is binded, and
- exactly one bound action defines `method`.

9.2 Qualified Calls

```
value.method::X();
```

Legal only if `X` is part of the value's bind or value-level composition. Qualification never introduces new semantics.

10 Function Parameters and Calling Conventions

Function parameters explicitly determine which semantic layer is preserved across the call boundary. Bliss supports four distinct parameter forms:

1. Data parameters (representation only)
2. Binded parameters (semantic identity preservation)
3. Action parameters (behavior-only polymorphism)
4. `with` parameters (representation + capability constraint)

10.1 Data Parameters (Semantic Erasure)

```
fx someStuff(Data x);
```

Rules:

- Accepts any binded type with underlying `Data`
- All actions and meaning are stripped at the boundary
- Only raw data is visible inside the function
- Implicit stripping may produce a compiler warning (equivalent to `ext`)

```
Data::A a;
Data::B b;
Data::(A, B) c;

someStuff(a); // valid (semantic erasure)
someStuff(b); // valid
someStuff(c); // valid
```

10.2 Binded Parameters (Semantic Preservation)

```
fx process(Data::A a);
```

Rules:

- The provided bind must contain at least the action set required by `A`
- Compatibility is defined by action-set inclusion
- Meaning associated with `A` is guaranteed inside the function
- No reinterpretation is allowed

If $S(T)$ denotes the action set of bind T, then:

$$\texttt{Data::T satisfies Data::A} \iff S(A) \subseteq S(T).$$

```
process(a); // valid
process(b); // invalid
process(c); // valid if c contains at least A's action set
```

10.3 Action Parameters (Behavioral Polymorphism)

```
fx doThing(x with ActionType);
```

Rules:

- Any binded type implementing `ActionType` is accepted

- Underlying data is inaccessible
- Only behavior defined in the action is visible
- No semantic identity is preserved

```
Data1::X a;
Data2::Y b;

// X binds ActionType with Data, Y does not
doThing(a); // valid
doThing(b); // invalid
```

10.4 `with` Parameters (Representation + Capability)

```
fx log(u8[][] str with (ActionGroup));
```

The `with` form defines a compound parameter type consisting of:

- an exact underlying data type, and
- a required action set.

Rules:

- The underlying data type must match exactly.
- The provided bind must contain the required action set.
- Both representation and capability are available inside the function.
- No specific interpretation token is required.

Formally, a value `Data::T` satisfies `D with (`A_1`, ..., `A_n`)` iff:

1. the underlying data type is exactly `D`, and
2. $\{A_1, \ldots, A_n\} \subseteq S(T)$.

This differs from `::`:

- `Data::A` requires a specific semantic interpretation.
- `Data with (A)` requires capability but not token identity.

The `with` construct does not create new semantic identity; it expresses a boundary constraint combining representation and behavior.

11 Explicit Semantic Projection

11.1 Data Extraction (`ext`)

`ext` explicitly destroys meaning and behavior, recovering raw data.

```
Data x = ext Data::(A, B, C, D);
```

Rules:

- All actions are removed.
- The result has open semantics.
- No meaning is preserved.

11.2 Reinterpretation After Extraction

```
Data::B y = (ext Data::A)::B;
```

Rules:

- Extraction always precedes reinterpretation.
- This is legal only because the intermediate value is raw data.
- Semantic changes are explicit and local.

12 Invalid Semantic Operations

The following are always illegal:

```
b::Y.length(); // semantic widening
b.length::Y(); // accessing non-existent semantics
```

Binded types define a closed semantic universe.

13 Reinterpretable Actions and Guidelines

Actions annotated with `@reinterpret` assign meaning to raw data.

```
@reinterpret
action UTF8 { }
```

Guidelines:

- Prefer using one reinterpretation at a time.
- Multiple reinterpretations may conflict.
- The compiler does not enforce semantic consistency.

Binded types exist to lock meaning and avoid ambiguity.

14 Type Aliasing and Qualified Names

When binded types are used heavily, fully qualified names can become cumbersome. Bliss therefore provides syntactic sugar for type aliasing:

```
using std.actions.summary;

data UserData {
   String name;
   int age;
}

bind summary, UserData as UserBinder {
   fx summarize(self) {
      return "Name: { self.name }\nAge: { self.age }"
   }
}

alias UserData::UserBinder as User; // dumb rename
alias UserData::UserBinder as SameButDifferent; // again dumb rename
```

15 Standalone Bindings

Sometimes a standalone type is desired without aliasing. Since all bindings are types as well, they may be used directly:

```
bind summary, UserData as User { ... } // implementations
```

Here `User` is the bound semantic type formed from `UserData` and the `summary` action.

16 Singleton Actions and Bindings

Some behaviors consist of a single action function that does not naturally belong to a larger conceptual group. Bliss provides a *singleton action* syntax as a purely syntactic convenience.

16.1 Singleton Action Definition

```
action fx Walkable.walk(self) -> void;
```

This is syntactic sugar for an action containing exactly one function. It introduces no new kind of action, and all semantic rules for regular actions apply unchanged.

16.2 Singleton Binding

```
bind Walkable.walk(self) {
    self.steps += 20;
}, DogData as WalkableDog;
```

This is semantically equivalent to a standard bind involving a single-action action type. The resulting type `WalkableDog` is a nominal binded (semantic) type with closed semantics.

16.3 Aliasing as a Unified View

```
alias DogData::WalkableDog as Dog;
```

This alias introduces no new meaning; it is only a convenient name for an existing semantic type.

16.4 Rationale

Singleton actions support single-responsibility behaviors without encouraging artificial abstraction. They reduce boilerplate, improve readability, and ease refactoring while remaining fully consistent with the Data–Action–Bind model.

17 Field Values

Fields inside a `data` definition may take different forms depending on how they are intended to be accessed and mutated. By default, all fields are readable and writable from any context. Bliss refines this behavior with field modifiers that control *visibility* and *mutability*:

- `hide`
- `private`
- `readonly`

Modifiers may be used individually or in combination.

17.1 `hide`

`hide` is the strongest field modifier. A `hide` field is visible only during construction of the data value and is completely inaccessible afterward: it is not readable or writable by actions and is not part of the observable semantic state. Such fields exist for runtime or native-layer mechanisms (handles, anchors, capability tokens) and are intentionally excluded from DAOP reasoning.

17.2 `private`

`private` restricts field visibility to actions only. A `private` field is invisible to user-level code but fully accessible within actions bound to the data; it remains part of the semantic state.

17.3 `readonly`

`readonly` restricts when a field may be written. A `readonly` field may be assigned only during construction and cannot be overwritten outside of actions, while remaining readable according to its visibility rules.

17.4 Combined Modifiers

Modifiers may be combined to further refine behavior. The combination `readonly hide` (or `hide readonly`) produces a field that can be assigned only during construction and cannot be mutated afterward, making it suitable for internal constants, cached metadata, or invariant state.

18 Canonical Summary

Data defines memory.
Actions define behavior.
Binded types define meaning.

Meaning is chosen, never inferred.