Working technical proposal

Quantified CUE

A Set-Theoretic Constraint Calculus

Polymorphic functions, dependent specifications,
and representation-independent modules

Aram Hăvărneanu

27 September 2026

Abstract. CUE descriptions are interpreted as predicates on joint environments of possible values. Existential projection explains instance formation and hidden witnesses; universal projection adds obligations over arbitrary values and semantic types. Both are adjoints to substitution, and both preserve refinement. Function types are universal predicates on an operational application relation. Their variance, intersection laws, and polymorphic instances follow from that interpretation. The proposal defines quantified declarations, legal scope extrusion, a conservative data-language embedding, and three language profiles. A worked catalogue covers calling conventions, higher-order programming, dependent invariants, and abstract modules. Existential modules use sealing and equality-respecting simulations to support representation-independent evolution. The reference model establishes refinement, function soundness, and representation-independent replacement under explicit hypotheses. A mandatory, resource-bounded static kernel checks an annotated polymorphic structural fragment. Every annotation is discharged by a static derivation. Explicit source assertions retain their runtime checks and contribute refinements to those derivations; annotation checking leaves executable code unchanged. Relevant consistency rejects explicit interfaces whose declared input regions have statically refuted successful results, independently of denotational inhabitation. Function arrows express partial correctness; constraint failure is permitted without making static typing optional. Impredicative quantification uses a single set-sized family of predicates.

Comparison baseline: CUE proposal 4484 and its theory addendum,
repository revision `a949162aedb8`, inspected 21 September 2026.

Contents

Conventions

Equations and proofs use the semantic reference calculus. Code listings use the proposed surface language and carry profile or extension labels. An expected contradiction is written `_|_`; an unresolved proof or evaluation obligation is written *incomplete*. A blocked static judgment is a compiler diagnostic, not the semantic value `_|_`. Relevant consistency is the static admissibility condition for explicit interface descriptions. Its failure blocks an interface even when the underlying function predicate is inhabited.

Notation	Meaning
V	A fixed set of operational inhabitants, including data and closures.
$\rho \in \mathrm{Env}(\Gamma)$	An assignment to the free variables of context Γ.
$[\![T]\!]_\rho \subseteq V$	The inhabitants of description T at ρ.
$P \leq Q$	Inclusion of predicates, hence refinement toward more information.
$\top, \bot$	The full subject set and the empty predicate.
$\mathcal{U} = \mathcal{P}(V)$	The set of semantic type interpretations.
$\mathcal{U}_1$	Universal-only, rank-1 surface profile.
$\mathcal{S}_1$	Symmetric rank-1 profile, with scoped existential packages.
$\mathcal{S}_H$	Full symmetric, impredicative higher-rank profile.
$\mathcal{D}$	The dependent-value extension of the indicated profile.
$\mathcal{A}$	The opaque-module abstraction discipline.
$\mathcal{K}$	The mandatory static-checking profile, independent of rank.
$\mathsf{Comp}(T)$	A computation with successful results in T, possibly failing.

Universal-only refers to the *new explicit type-binding surface*. Ordinary CUE instance variables retain their existential completion semantics in all three profiles. Rank restrictions concern type quantifiers; dependence on ordinary values is an independent design axis.

The forms `forall (A, B) T` and `exists A { ... }` bind variables in a description. A quantified field `f(A, B): T` constrains one subject at every instance. A parametric alias `S(A) = T` abbreviates a description by capture-avoiding substitution; `S(U)` applies the alias. Instantiation selection `f[U]` refers to the same subject at a universal instance.

The comparison uses proposal 4484 and its addendum at the cited repository revision [5, 6]. Stable CUE expressions retain their existing elaboration and observation rules.

Chapter 1

The existing constraint semantics

1.1 Descriptions and completions

CUE orders descriptions by subsumption and defines unification as their greatest lower bound. At the level of successful data completions, the corresponding set interpretation is inclusion, intersection, and union [1].

Fix a set D of complete data inhabitants. A primitive description denotes a subset of D; a concrete atom denotes a singleton. Write

$$[\![1]\!] = \{1\}, \qquad [\![\texttt{int}]\!] = \mathbb{Z}, \qquad [\![S \,\&\, T]\!] = [\![S]\!] \cap [\![T]\!], \qquad [\![S \mid T]\!] = [\![S]\!] \cup [\![T]\!].$$

An open record denotes the records satisfying its field predicates; a closed record additionally constrains its permitted labels. Field absence is a separate possibility from a present field with an unconstrained value.

An open declaration is more accurately a relation than a product of its field marginals. For example:

```
config: {
    x: int
    y: x + 1
}
```

has the relational interpretation

$$R(x, y) \equiv x \in \mathbb{Z} \ \wedge \ y = x + 1.$$

Its two fields cannot be interpreted as independent choices of integers. Projection onto an exported interface hides the other variables existentially. A complete exported record retains its exposed fields as the subject being constrained.

1.2 The sense in which variability is existential

A field declaration `x: int` leaves a witness to be supplied by a completion. Adding `x: >=0` removes negative witnesses. It imposes no obligation to produce an implementation working uniformly for every integer.

More generally, if Δ names local witnesses and Γ the exposed interface, the description of that interface is

$$[\![\exists \Delta \, P]\!] = \{\rho \in \mathrm{Env}(\Gamma) \mid \exists \delta \in \mathrm{Env}(\Delta).\ P(\rho, \delta)\}.$$

This is the existential reading of ordinary CUE variability. Built-in list and pattern constraints may themselves validate all matching elements; a general binder for an arbitrary semantic type is an additional facility.

Sharing precedes projection. Two contributions to one field elaborate to $\exists x\,(P(x) \wedge Q(x))$. Two independently introduced witnesses elaborate to $(\exists x\, P(x)) \wedge (\exists y\, Q(y))$. These formulas carry different sharing information.

Example 1.1 — Independent instantiations of a description. *Existing CUE*

```
#Cell: {value: int}
left:  #Cell & {value: 1}
right: #Cell & {value: 2}
```

Each instantiation has its own field cells. Within each cell, repeated constraints share that cell. CUE references copy the associated expression and rebind internal references to the corresponding copies [2]. The reference calculus therefore receives an elaborated graph with explicit sharing, rather than equating every textual identifier with one global logic variable.

1.3 Universal implementation obligations

A generic transformation needs one implementation whose behavior is valid for arbitrary types. Write $\mathrm{Map}(A, B, m)$ for the requirement that m applies a supplied transformation from A to B elementwise to a finite A-list and returns the corresponding B-list. The two statements

$$\exists A, B, m.\ \mathrm{Map}(A, B, m), \qquad \exists m.\ \forall A, B.\ \mathrm{Map}(A, B, m)$$

are different specifications. The first asks for some usable instance. The second fixes an implementation and requires every instance. In particular, existentially choosing $A = \varnothing$ can trivialize an input obligation.

This distinction applies to records as well as functions: a reusable module may contain one operation that works at every element type, and a hidden representation shared by several operations. The language needs to retain both kinds of binding and their dependency order.

1.4 The semantic layer of existing features

The reference model is the *completion layer*. The implementation also retains closure provenance, scope graphs, field-presence information, and preference information. A default-bearing description has successful completions and a preferred subset. Default selection is an observation of that additional structure. Comprehensions elaborate constraints using the host language's established evaluation rules [3].

Legacy programs retain their data observations and evaluation rules. Refinement theorems apply to a fixed elaborated predicate and a fixed quantifier dependency structure.

Chapter 2

Quantification as a set-theoretic operation

2.1 Contexts and subjects

Definition 2.1 (Predicate fiber)**.** *For a context* Γ *with set of assignments* $\mathrm{Env}(\Gamma)$*, let* $\mathcal{P}_\Gamma = \mathcal{P}(\mathrm{Env}(\Gamma))$*, ordered by inclusion. A type expression with subject* z *denotes a predicate in* $\mathcal{P}_{\Gamma,z:V}$*.*

The subject is the value described by an expression. A quantifier binds its listed variables and leaves that subject fixed. A record field supplies a projection of the enclosing record as the subject of its field expression.

A substitution $h : \mathrm{Env}(\Delta) \to \mathrm{Env}(\Gamma)$ induces reindexing $h^*P = h^{-1}(P)$. It preserves all set unions and intersections. For a fixed sort S, let $\pi : \mathrm{Env}(\Gamma) \times S \to \mathrm{Env}(\Gamma)$ be the projection. Define

$$\exists_\pi P = \{\rho \mid \exists s \in S.\ (\rho, s) \in P\}, \tag{2.1}$$

$$\forall_\pi P = \{\rho \mid \forall s \in S.\ (\rho, s) \in P\}. \tag{2.2}$$

These are predicates on the original interface, not families left outside the semantic domain.

Proposition 2.2 (Adjunctions)**.** *For predicates* P *on the extended context and* Q *on the base context,*

$$\exists_\pi P \subseteq Q \iff P \subseteq \pi^* Q, \qquad \pi^* Q \subseteq P \iff Q \subseteq \forall_\pi P.$$

Proof. The first equivalence expands to: every $(\rho, s) \in P$ has $\rho \in Q$. The second expands to: for every $\rho \in Q$ and every $s \in S$, $(\rho, s) \in P$. □

Thus $\exists_\pi \dashv \pi^* \dashv \forall_\pi$. This is the subset hyperdoctrine underlying the proposal [7, 8].

2.2 A single impredicative type sort

Fix a set V of operational inhabitants, including a designated set of closure codes with captured environments. Their application relation is fixed independently of type interpretation. Take

$$\mathcal{U} = \mathcal{P}(V), \qquad \rho(A) \in \mathcal{U}.$$

The sort Type classifies predicates on V. It is a syntactic sort, not an inhabitant of V satisfying $\mathsf{Type} : \mathsf{Type}$. Type operators are interpreted pointwise in a valuation:

$$[\![A]\!]_\rho = \rho(A), \qquad [\![T \wedge U]\!]_\rho = [\![T]\!]_\rho \cap [\![U]\!]_\rho, \qquad [\![T \vee U]\!]_\rho = [\![T]\!]_\rho \cup [\![U]\!]_\rho.$$

Quantification has the same result sort:

$$[\![\forall A\, T]\!]_\rho = \bigcap_{S \in \mathcal{U}} [\![T]\!]_{\rho[A:=S]}, \tag{2.3}$$

$$[\![\exists A\, T]\!]_\rho = \bigcup_{S \in \mathcal{U}} [\![T]\!]_{\rho[A:=S]}. \tag{2.4}$$

Both expressions denote subsets of V, hence members of $\mathcal{U}$. A variable may therefore be instantiated by the interpretation of a quantified type. This is impredicative polymorphism. Interpretation is defined by structural recursion on the finite type expression; quantification ranges over a previously fixed set of predicates, not over recursive evaluations of every source type.

Proposition 2.3 (Impredicative substitution)**.** *For capture-avoiding substitution of any well-sorted type S, including a quantified type,*

$$[\![T[S/A]]\!]_\rho = [\![T]\!]_{\rho[A:=[\![S]\!]_\rho]}.$$

Proof. Induct on T. Boolean and structural constructors commute with substitution. At a fresh quantifier, the two sides take the same set-indexed intersection or union over $\mathcal{U}$. The arrow case uses the fixed operational application relation. □

Operational arrows are predicates on a fixed set of realizers, rather than the full set-theoretic exponentials B^A. This is the relevant departure from the full-function-space interpretation obstructed by Reynolds's theorem [9]. Interpreting erased quantification by intersection has precedents in realizability models of System F [27]. Allowing failure changes the arrow predicate; it does not by itself justify impredicativity. The justification here is the fixed operational carrier and closure of $\mathcal{P}(V)$ under the displayed operations.

A smaller Henkin family may replace $\mathcal{P}(V)$ if it is closed under every type constructor and quantified interpretation admitted by the source fragment. The powerset is a simple reference choice, not a representation to enumerate. An implementation stores finite syntax and derivations. Data descriptions are interpreted through their completion predicates; refinement of a description is inclusion of those predicates. Type binders remain erased, so the predicate sort cannot be reflected or used to construct a runtime self-membership test.

2.3 Bounds and dependent ranges

A type bound $A : T$ abbreviates the range

$$\mathrm{Sub}(T, \rho) = \{S \in \mathcal{U} \mid S \subseteq [\![T]\!]_\rho\}.$$

A value binder x in T ranges over $[\![T]\!]_\rho$. Bounds may depend on earlier variables. Their context extension is the comprehension set

$$\Gamma.D = \{(\rho, d) \mid \rho \in \mathrm{Env}(\Gamma),\ d \in D_\rho\}.$$

Equations (2.1) and (2.2) then quantify over the fiber D_ρ. This is the same construction for constant and dependent ranges.

For fixed ambient sorts, bounded quantification can be expressed by guards:

$$\exists a \in D\ P = \exists a\ (D(a) \wedge P), \qquad \forall a \in D\ P = \forall a\ (D(a) \Rightarrow P).$$

The implication is a semantic operation in the Boolean predicate model; a surface implementation can retain the guarded clause directly.

2.4 Substitution and duality

For a pullback square of context maps, reindexing commutes with both quantifiers:

$$h^*(\exists_\pi P) = \exists_{\pi'}(h'^* P), \qquad h^*(\forall_\pi P) = \forall_{\pi'}(h'^* P).$$

To verify either equation, fix the base assignment and compare the corresponding fibers of the pullback. Their witnesses are in bijection. These Beck–Chevalley laws govern capture-avoiding copying of quantified descriptions.

In each Boolean fiber,

$$\forall_\pi P = \neg\exists_\pi(\neg P).$$

The semantic duality is available even in a surface fragment that exposes only positive conjunctions, disjunctions, and universal binders.

2.5 Empty ranges and uniform subjects

The empty-index conventions are essential:

$$\bigcap \varnothing = V, \qquad \bigcup \varnothing = \varnothing.$$

Hence $\forall a \in \varnothing\ P = \top$ and $\exists a \in \varnothing\ P = \bot$. If $\mathcal{U}$ contains bottom, then $\forall A\ A = \bot$. If lists are finite and bottom is admissible, $\forall A\ \mathsf{List}(A)$ consists exactly of the empty list.

Example 2.1 — One subject at every instance. $\mathcal{U}_1$

```
empty(A): [...A]                // []
impossible(A): A                // _|_
impossibleRecord(n in 0 | 1): {
    value: n                    // one field cannot be both 0 and 1
}
```

These are intersections of predicates on one list, value, or record. A parametric alias abbreviates a description by substitution. A descriptor-producing function instead returns a description as a value.

Chapter 3

Unification, floating, and monotonicity

3.1 Pointwise unification

For a common range D,

$$(\forall a \in D\, P(a)) \wedge (\forall a \in D\, Q(a)) = \forall a \in D\, (P(a) \wedge Q(a)). \tag{3.1}$$

This follows by regrouping intersections. Binder names are alpha-equivalent.

Example 3.1 — Refining a polymorphic record across declarations. $\mathcal{U}_1$

```
r(A): {wrap: func(A) -> {value: A}}
r(B): {wrap: func(B) -> {tag: "wrapped"}}

// Unified description, after alpha-renaming:
r(T): {wrap: func(T) -> {value: T, tag: "wrapped"}}
```

The two output obligations constrain one operation at each T.

Equal spelling does not identify independent existential witnesses:

$$(\exists a\, P(a)) \wedge (\exists b\, Q(b)) = \exists a, b\, (P(a) \wedge Q(b)).$$

Shared existential witnesses are introduced by a shared scope and are conjoined before their binder is discharged.

3.2 Different bounds retain different obligations

For a common body P,

$$(\forall a \in D\, P(a)) \wedge (\forall a \in E\, P(a)) = \forall a \in D \cup E\, P(a).$$

Here $D \cup E$ is a union of *assignments*. For type bounds,

$$\mathrm{Sub}(B) \cup \mathrm{Sub}(C) \subseteq \mathrm{Sub}(B \cup C)$$

can be strict: mixed subtypes of $B \cup C$ need lie below neither arm.

Example 3.2 — Two bounds do not become one upper bound. $\mathcal{U}_1$

```
choose(A: int):    func(A, A) -> A
choose(A: string): func(A, A) -> A

// The stronger declaration also covers mixed unions:
stronger(A: int | string): func(A, A) -> A
```

The first pair retains both guards. With different bodies, its meaning is $\forall A\, ((A \subseteq \mathbb{Z} \Rightarrow P(A)) \wedge (A \subseteq \mathsf{String} \Rightarrow Q(A)))$.

3.3 Legal scope extrusion

For $a \notin \mathrm{FV}(Q)$, existential Frobenius reciprocity gives

$$(\exists a\, P) \wedge Q = \exists a\, (P \wedge Q).$$

For a nonempty fixed range, the universal counterpart holds:

$$(\forall a\, P) \wedge Q = \forall a\, (P \wedge Q).$$

For a possibly empty dependent range, the universally quantified expression must retain Q outside the guard. Implementations store the guard and the independent conjunct separately until inhabitation is known.

Example 3.3 — An ordinary field floats as a constant obligation. $\mathcal{U}_1$

```
r(A): {id: func(A) -> A}
r:    {label: "utilities"}

// The type sort contains the empty predicate:
r(A): {id: func(A) -> A, label: "utilities"}
```

Independent same-kind quantifiers commute. An existential and a universal commute only when a separately justified independence law applies. In general,

$$\exists y\, \forall x\, (y = x) \quad \neq \quad \forall x\, \exists y\, (y = x).$$

The dependency graph records which universal choices a witness may depend on. File order is irrelevant; dependency order is semantic.

3.4 Disjunction and projection

Existential projection preserves unions; universal projection preserves intersections. Their opposite distributivities fail.

Example 3.4 — Binder scope across a union. $\mathcal{U}_1+\mathcal{D}$

```
a: forall (n in 0 | 1) (n | (1 - n))  // 0 | 1
b: (forall (n in 0 | 1) n) |
   (forall (n in 0 | 1) (1 - n))      // _|_
```

Projection of an entire record must also retain joint witnesses. Take $R_0 = \{(s, 0)\}$ and $R_1 = \{(s, 1)\}$. Projection p onto the first coordinate satisfies

$$p(R_0 \cap R_1) = \varnothing, \qquad p(R_0) \cap p(R_1) = \{s\}.$$

Pushing record selection through a universal intersection is therefore only an inclusion in general. Pointwise unification operates on the whole subject, including its correlated fields.

3.5 Refinement theorem

Theorem 3.1 (Monotone quantified refinement)**.** *Let $P' \subseteq P$ be predicates on one elaborated context, with the same quantifier dependency graph. Every legal composite Q of existential and universal projections satisfies $Q(P') \subseteq Q(P)$. In particular, $Q(P \wedge R) \subseteq Q(P)$.*

Proof. Both quantifiers are monotone, either by proposition 2.2 or directly from their membership clauses. Composition preserves monotonicity. □

A contradictory closed predicate remains contradictory. A projected atomic result constrained to $\{v\}$ can remain $\{v\}$ or become empty. Its successful completion cannot change to a different atom.

3.6 Refinable bounds are correlated variables

Let an existential semantic type X have upper bound Number, and let

$$P(X, f) \equiv X \subseteq \mathsf{Number} \ \wedge \ \forall A \subseteq X \, T(A, f).$$

Adding $X \subseteq \mathbb{Z}$ gives $P' = P \wedge (X \subseteq \mathbb{Z}) \subseteq P$. The result remains monotone after existentially hiding X. The universal obligation for one fixed X is contravariant in its bound; the *joint predicate on possible X and f* is refined by conjunction. A solver retains X and its correlations. It does not substitute its current upper approximation into every occurrence.

Example 3.5 — A particular client and a universally capable producer. $\mathcal{U}_1$

```
job: {
    x: number
    f: func(x) -> string
    result: f(x)
}
job: {x: int & >=0}
job: {x: 7, f: func(n: int) -> string: "\(n)"}
```

In a complete instance, `x` denotes one integer and the function obligation uses its singleton type. The joint relation retains that dependency. A separate `func(number) -> string` requires all numeric inputs. **Implementation obligation.** The elaborator distinguishes an ordinary witness reference, whose type-position use is a singleton, from a type-sorted variable, whose use is its decoded semantic type. Copying a CUE description preserves that classification and the copied witness graph.

Chapter 4

Function types from universal obligations

4.1 Values, computations, and failure

A closure is an operational value with code, a captured environment, and a fixed argument-binding protocol. Let $F \subseteq V$ be the set of closures. A call packet records positional arguments, labels, and omission. A complete packet contains values; a symbolic packet denotes a relation on its possible complete packets, with sharing retained.

Write $\mathrm{run}(f, p) \Downarrow v$ for a successful complete return. Constraint failure, domain rejection, and an allowed execution that does not return have no successful result. A symbolic successful result describes a set S of complete returns. Bottom is the empty completion predicate; it is not an extra successful value in every type.

For a result predicate B, define

$$H_B(f, p) \equiv \forall v.\ (\mathrm{run}(f, p) \Downarrow v \Rightarrow v \in B).$$

The function type is

$$A \Rightarrow B = \{f \in F \mid \forall p \in A.\ H_B(f, p)\}. \tag{4.1}$$

Thus $f : A \Rightarrow B$ constrains successful returns; it promises neither a return nor success. On symbolic descriptions the analogous condition is $S \subseteq B$, including $S = \varnothing$. Ordinary arrows admit CUE constraint failure without an effect annotation. A recursion policy remains an independent execution restriction.

A computation judgment $e : \mathsf{Comp}(T)$ means that every successful complete result of e belongs to T. A value judgment $v : T$ asserts actual membership. Only the latter supplies a witness for an existential or a value for a completed field. In particular, a failing computation can have $\mathsf{Comp}(\bot)$ while no returned value inhabits $\bot$.

A proved empty successful-result predicate reduces the corresponding computation to $\_ \mid \_$. A required data field with that predicate makes its containing configuration inconsistent, even before a concrete witness is supplied. Thus `x: int` conjoined with `x: bool` reduces to bottom. Suspension is justified by an unresolved predicate, not by retaining a predicate already proved empty.

A closure is itself an operational value. A never-successful closure can inhabit $A \Rightarrow \bot$, although applying it supplies no returned value. The empty-domain identity $\varnothing \Rightarrow B = F$ also holds. Emptiness of a result predicate, emptiness of a set of closures, and failure of a particular call are therefore distinct judgments. Their scopes govern bottom propagation in section 11.8. Explicit interfaces also undergo relevant-consistency checking: a refuted successful result on a non-refuted stated input region is a static interface error. This condition does not change equation (4.1); it restricts the descriptions admitted at source boundaries.

The static judgment in chapter 11 checks the source body and calling convention by fixed rules. An annotation creates a proof obligation. It creates no result filter or other executable operation.

4.2 Variance and intersections

Proposition 4.1 (Arrow laws)**.** *For all domain and result sets:*

$$A \subseteq C,\ D \subseteq B \Longrightarrow (C \Rightarrow D) \subseteq (A \Rightarrow B),$$
$$(A \cup C) \Rightarrow B = (A \Rightarrow B) \cap (C \Rightarrow B),$$
$$A \Rightarrow (B \cap D) = (A \Rightarrow B) \cap (A \Rightarrow D).$$

Proof. The first two statements restrict or split the universal input obligation. For the third, every successful result must belong to both B and D. Intersecting those obligations is exactly the obligation for $B \cap D$. Failure satisfies each implication vacuously. The identities also hold for a nondeterministic relation when every successful result is constrained. □

Thus $(\mathbb{Z} \Rightarrow \mathsf{String}) \cap (\mathsf{Number} \Rightarrow \mathsf{String}) = \mathsf{Number} \Rightarrow \mathsf{String}$. More generally, an intersection of arrows preserves input/output correlations that a single arrow may lose.

Example 4.1 — A correlation-preserving overloaded contract. $\mathcal{U}_1$

```
convert: (func(int) -> string) & (func(string) -> int)
```

One implementation must meet both obligations. A body may perform explicit case analysis. The type intersection itself performs no branch selection. On overlapping domains, all result obligations apply.

4.3 Implementations and application refinement

For a complete closure f, unification with T denotes $\{f\} \cap [\![T]\!]$. A successful conformance check preserves the closure. An incompatibility makes that intersection empty. A symbolic closure retains its captured-environment predicates. Executable use additionally requires the mandatory static certificate defined in chapter 11. An unproved strict conformance judgment is blocked rather than retained as a runtime type test.

For a set of closures X and input packets Q, define

$$\mathrm{App}(X, Q) = \{v \mid f \in X,\ p \in Q,\ \mathrm{run}(f, p) \Downarrow v\}.$$

Joint environments are retained when X and Q share variables. Then

$$\mathrm{App}(X \cap Y, Q) \subseteq \mathrm{App}(X, Q) \cap \mathrm{App}(Y, Q). \tag{4.2}$$

Membership on the left has one closure witness that works on both sides. Equality can fail: distinct closures can agree at a particular input while having empty singleton intersection.

Example 4.2 — Capabilities and one-call constraints. $\mathcal{U}_1$

```
wide:   func(number) -> string
wide:   func(x: int) -> string: "ok"  // blocked: static domain mismatch

narrow: func(int) -> string
narrow: func(x: number) -> string: "ok" // conforming implementation
```

The second attachment satisfies the static contravariant interface rule. The first is blocked by that rule even though a narrow implementation that fails on other inputs could satisfy the wider partial-correctness predicate. Semantic membership is intentionally more permissive than source certification. A particular invocation is constrained with existential argument cells as in chapter 3.

For one declaration subject, collect all interface conjuncts before checking its available body. An inline result annotation and an independently declared function type contribute to the same predicate. The checker retains the body's inferred refinements or its proof graph; a weaker previously published result annotation does not discard that evidence. Interface checking never modifies that body. Section 11.7 gives the normalization rule and its consequences for declaration regrouping.

4.4 Erased polymorphic functions and unification

A polymorphic identity constraint denotes

$$I = \bigcap_{A \in \mathcal{U}} (A \Rightarrow A).$$

Singleton instantiation proves that a successful return on x is x. A member of I may fail on some inputs. Thus I describes partial identities.

Example 4.3 — Universal instances and explicit relevant observations. $\mathcal{U}_1+\mathcal{K}$

```
id(A): func(A) -> A
id:    func(int) -> int       // redundant instance

quiet(A): func(A) -> A
quiet:    func(int) -> bool   // invalid explicit interface on int
```

The ordinary integer clause supplies a relevant input region. Instantiating the universal clause at that region gives both $\mathbb{Z}$ and Bool as successful-result requirements. Their intersection is empty, so the accumulated explicit interface is rejected without a body or a call. The source condition and its finite instantiation rule are defined in section 11.5.

Denotationally the conjunction remains inhabited. For example,

$$q = \lambda x.\,(x \;\&\; \mathsf{Bool}) \quad\Longrightarrow\quad q \in I \cap (\mathbb{Z} \Rightarrow \mathsf{Bool}), \qquad q(\mathsf{true}) = \mathsf{true}.$$

Behavior outside the stated integer region cannot repair its refuted successful observations. Relevant consistency diagnoses the explicit specification; it does not assert that the set of partial identities is empty. The predicate $I \cap (\mathbb{Z} \Rightarrow \mathsf{Bool})$ remains available to semantic reasoning and to internal descriptions of failing applications.

Example 4.4 — An empty value hypothesis. $\mathcal{U}_1$

```
x: int
x: bool                      // _|_, without a supplied value
uninhabited(A): A             // _|_, by instantiation at bottom
```

These predicates have no value inhabitants. They differ from an inhabited set of closures whose calls fail on a specified domain.

The meet operator is a refinement-preserving primitive. If d and e denote completion predicates D and E, their successful unification denotes $D \cap E$. Hence

$$D \subseteq A,\; E \subseteq B \quad\Longrightarrow\quad D \cap E \subseteq A \cap B.$$

No compatibility or inhabitance premise is needed. This gives the primitive partial type $\forall A, B.\ (A \times B) \Rightarrow (A \cap B)$. Symbolic operands retain their joint constraints; a record expression with additional permitted fields is not identified with an exact ground record.

Example 4.5 — A language operator with a polymorphic type. $\mathcal{U}_1+\mathcal{K}$

```
f(A): func(xs: [A, A]) -> A: xs[0] & xs[1]
x: f([{a: 1}, {b: 1}])       // {a: 1, b: 1}
y: f([1, 2])                 // _|_: conflicting concrete operands

meet(A, B): func(x: A, y: B) -> (A & B): x & y
z: meet({a: 1}, {b: true})   // {a: 1, b: true}
```

The body of `f` synthesizes $A \cap A = A$, so certification is local. For `x`, instantiate A at `{a: 1} | {b: 1}`. The returned record refines both open descriptions and therefore their union. Its more precise fields follow from evaluating the body, not from a claim that an arbitrary inhabitant of the signature always performs a meet. The incompatible call `y` is refuted when its concrete operands expose the conflict.

The explicit result $A \cap B$ of `meet` is not refuted under its rigid declaration variables. It therefore passes relevant consistency. A generated instance such as $\mathbb{Z} \cap \mathsf{Bool}$ may describe computation failure without creating a new explicit interface obligation.

4.5 Effects and conditional guarantees

Ordinary arrows already allow constraint failure. An optional effect row records observable foreign errors or external actions beyond that failure. For an error set ϵ, define

$$A \Rightarrow_\epsilon B = \{f \in A \Rightarrow B \mid \forall p \in A, e.\ (\mathrm{run}(f, p) = \mathsf{Raise}(e) \Rightarrow e \in \epsilon)\}.$$

This is still a partial-correctness contract. A bridge can expose `func(int) -> int` when conversion failure becomes bottom, or `func(int) -> int !bridge` when its error tags are observable. Neither signature needs native machine widths. Registration must still prove logical parameter, result, and protocol compatibility in the strict kernel. An integer adapter cannot be certified as a number adapter merely by rejecting fractions at runtime.

Termination, guaranteed success, and absence of a selected effect are stronger properties with separate proof obligations. A dependent law that actually executes a call as a required field can impose such an obligation locally; an ordinary arrow never supplies it implicitly.

Chapter 5

Surface binding and conservative extension

5.1 Quantifier expressions

The core surface expressions are

```
forall (A, B) T
exists A { ... }
forall (A: number) func(A, A) -> [A, A]
forall (n in int & >=0) T
```

Parentheses group multiple binders or a constrained binder; a single simple binder may omit them. The body is the following expression. A record body is already delimited by braces. Quantifiers have low expression precedence, and parentheses delimit narrower scopes inside unions and arrows.

Unadorned A has sort Type. `A: T` bounds that semantic type by inclusion. `x in T` binds an ordinary value. The explicit form `A in Type: T` records the type sort and its upper bound. All binders are lexical and alpha-renamable. Bounds may reference earlier binders and outer variables.

5.2 Quantified field declarations

The declaration

```
map(A, B): func(func(A) -> B, [...A]) -> [...B]
```

elaborates exactly to

```
map: forall (A, B) func(func(A) -> B, [...A]) -> [...B]
```

The field value is fixed outside the quantifier. The outer record's field selection is retained as the subject of the quantified predicate.

For a quantified record:

```
r(A): {
    id:   func(A) -> A
    wrap: func(A) -> {value: A}
}
```

the order is $\exists r\, \forall A\, P(r, A)$ when the surrounding record is closed existentially. Both projections refer to the same record for all A. The field-sugar form is concise for rank-1 declarations; RHS syntax also permits nested and higher-rank positions.

5.3 Quantifier blocks

A lexical binder prefix can be written in a record block:

```
x: {
    exists T: U
    forall V: W
    // The whole following declaration body is in this prefix.
    value: T
    transform: func(V) -> V
}
```

This elaborates to

```
x: exists (T: U) forall (V: W) {
    value: T
    transform: func(V) -> V
}
```

The ordered prefix records dependency. The declarations after the prefix are conjoined without source-order dependence. Contributions from other files unify with the entire quantified subject; they do not capture these private binder names merely by spelling them alike. A named interface may explicitly export a sharing equation when such sharing is intended.

A syntactically asymmetric alternative requires external parentheses for universals and block prefixes for existentials:

```
x(V: W): {
    exists T: U
    value: T
    transform: func(V) -> V
}
```

Its prefix is $\forall V\, \exists T$, rather than $\exists T\, \forall V$. The two spellings express different dependence. Both quantifiers still use the dual semantic operations of chapter 2.

5.4 Function-local generics and parametric aliases

A compact function-only spelling is possible:

```
map: func<A, B>(func(A) -> B, [...A]) -> [...B]
```

It expands to an outer universal on the function subject.

A parametric alias binds parameters in a description:

```
Box(A) = {value: A}
Pair(A, B) = [A, B]
```

The = form defines an alias; the `:` form constrains a field. Alias application substitutes its arguments capture-avoidingly into the body, so `Box(int)` expands to `{value: int}`. Alias parameters use the same sort and bound syntax as quantified binders; arguments must satisfy those bounds. A nonparametric alias uses the existing form `let S = T`.

In contrast, `box(A): {value: A}` requires one value lying in every A and is bottom when bottom is admissible. Reified descriptor functions retain their descriptor arguments as correlated variables.

`foo[T]` selects the instance of a universally constrained subject justified by substituting `T` for its binder. For a universal description, selection refers to the same inhabitant at that instance; it retains the subject and its universal constraints. Thus `id[int](3)` uses the integer instance of `id`. Selection is erased. Implicit instantiation uses fresh flexible variables at the call site and retains the universal declaration.

5.5 A conservative embedding

Definition 5.1 (Legacy-data observation)**.** *A legacy-data observation is evaluation and export using the pre-extension operators and data formats, applied to a program in the legacy syntax. Its completion domain is D, embedded into the extended V by $i : D \hookrightarrow V$.*

Theorem 5.2 (Relative conservativity)**.** *Suppose the extension retains the legacy elaborator, base evaluation rules, default-selection rules, and data constructors. For every legacy description T and legacy assignment ρ,*

$$i^{-1}([\![T]\!]^{+}_{i\rho}) = [\![T]\!]^{0}_{\rho}.$$

Closed legacy programs have the same legacy-data observations under the extended evaluator.

Proof. For data predicates, the claim is the defining restriction of each extended primitive to D. Inverse image preserves intersections and unions. Record and list constructors preserve the embedding and the field-sharing elaboration. Existing local projection and copying operations are unchanged. The operational statement follows by induction on the unchanged evaluation derivation, with identical default selection and export. □

The theorem is observational and data-relative. Extended top additionally admits new kinds of inhabitants in new-language contexts. This distinguishes conservativity from equality of global carriers.

Design rule. Gate new binder and arrow forms by language version or experiment attributes. Treat `forall`, `exists`, and `func` as contextual keywords in the selected syntax. Existing data files retain their parsing, closing, copying, and evaluation behavior.

Existing struct-as-function code remains a relational program:

```
#Add: {a: int, b: int, out: a + b}
sum: (#Add & {a: 1, b: 2}).out
```

A function declaration adds a reusable universal obligation and a concise call form. Refactoring to that declaration is an explicit program change, checked against the resulting contract.

Chapter 6

Three implementation profiles

6.1 Rank, instantiation, and interface boundaries

Type rank measures the placement of polymorphic binders relative to function arguments. A callback required to work at all types is higher-rank; an ordinary higher-order function such as `map` can have a rank-1 type. The semantic sort Type is impredicative. The rank-1 source profiles limit inferred and explicit instance types to quantifier-free bodies at executable boundaries; $\mathcal{S}_H$ permits explicitly supplied quantified instances. This is a source formation restriction, not a hierarchy of semantic universes.

For the restricted profiles, use a *boundary-prenex* discipline. Within one executable signature clause, type quantifiers occur in an outer prefix over a quantifier-free body, after justified normalization. Module interfaces are explicit additional boundaries. Finite intersections retain several prefixed clauses when their bounds or dependencies differ. An alias expands for the rank check; naming a higher-rank type does not change its rank.

A positive universal may sometimes float through a result arrow when its variable is absent from the input and the established deterministic arrow law justifies the move. Quantifiers over argument types retain their position. Existentials whose witnesses depend on arguments likewise retain that dependence.

6.2 Full symmetric, higher-rank CUE: $\mathcal{S}_H$

$\mathcal{S}_H$ admits both quantifiers at arbitrary well-sorted type positions, with the single impredicative type sort. It includes first-class existential packages, polymorphic arguments, and nested quantified records.

```
use: func(forall A func(A) -> A) -> [int, string]

#Showable: exists A {
    value: A
    show:  func(A) -> string
}
show: func(#Showable) -> string
```

The checker uses explicit introduction and elimination rules, scope levels, and annotations. Higher-rank bidirectional checking provides a useful proof-theoretic organization [20], while Boolean connectives and dependent predicates require their own sound constraint procedures.

Example 6.1 — An explicitly impredicative instance. $\mathcal{S}_H{+}\mathcal{K}$

```
let Identity = forall A func(A) -> A
id(A): func(x: A) -> A: x
again: id[Identity](id)
a: again[int](3)
b: again[string]("three")
```

The instance argument is itself universally quantified. Checking substitutes finite type syntax; it does not search for an impredicative instance or evaluate any type representation at runtime. Implicit higher-rank inference may request this annotation. The mandatory checking profile in chapter 11 is orthogonal to the three formation profiles.

6.3 Symmetric, rank-1 CUE: $\mathcal{S}_1$

$\mathcal{S}_1$ admits existential and universal prefixes at declaration and module-interface boundaries. An executable callback parameter is monomorphic at each call. Abstract modules are opened at their module boundary, exposing a rigid local representation name and monomorphic operations to their clients. Passing an inline quantified package to a function, or receiving a universally quantified callback, uses $\mathcal{S}_H$.

A representative interface is

```
Counter: exists State {
    zero: State
    next: func(State) -> State
    read: func(State) -> int
}
```

Module-level client code opens `Counter`, then passes `next` and `read` as ordinary monomorphic functions. The prefix can contain both kinds of binder, with explicit dependency order. Symmetry concerns their availability and semantic laws at the same boundaries; it does not authorize commuting them.

This profile supports ML-style abstract module reuse with a boundary-prenex rank restriction.

6.4 Universal-only, rank-1 CUE: $\mathcal{U}_1$

$\mathcal{U}_1$ adds only explicit outer universal type binders. Existing instance variables, local field cells, and call-instantiation variables remain existentially interpreted. The surface need not expose existential type packaging to express identity, mapping, composition, filtering, folds supplied as primitives, and many generic adapter interfaces.

The same semantics still defines existential projection internally. The implementation simply exposes a smaller formation and elimination interface. It therefore gains universal capability obligations and compositional polymorphic subtyping without requiring an opaque-module boundary in its first release.

```
id(A): func(A) -> A
compose(A, B, C): func(func(B) -> C, func(A) -> B) ->
    func(A) -> C
map(A, B): func(func(A) -> B, [...A]) -> [...B]
```

These declarations require rank-1 universals. They already distinguish reusable implementations from existentially constrained calls.

6.5 Coverage of host-language interfaces

Rank-1 signatures describe ordinary generic APIs. Higher-rank callbacks, abstract packages, and some lifetime-polymorphic or dependent APIs require a richer representation. GHC's documented distinction between rank-1 schemes and polymorphic parameters is a direct example [23]. A host binding must either express its actual quantifier structure or export a deliberately weaker interface with that loss recorded.

Facility	$\mathcal{U}_1$	$\mathcal{S}_1$	$\mathcal{S}_H$
Outer polymorphic arrows and records	yes	yes	yes
Finite intersections of scheme clauses	yes	yes	yes
Existential instance variables	yes	yes	yes
Explicit abstract module type witnesses	—	yes	yes
Polymorphic callback parameters	—	—	yes
First-class nested quantified packages	—	—	yes
Value dependence as a separate extension	$\mathcal{D}$	$\mathcal{D}$	$\mathcal{D}$

6.6 Implementation choice

$\mathcal{U}_1$ supports rank-1 generic interfaces. $\mathcal{S}_1$ adds explicit modular abstraction at stable boundaries. $\mathcal{S}_H$ admits quantified descriptions at arbitrary well-sorted positions. The inclusion of source fragments is conservative on programs already admitted by the smaller profile. A rank restriction does not by itself decide arbitrary semantic subtyping, quantified arithmetic, or program equivalence; each checker specifies its supported predicate fragment.

Chapter 7

Programs and calling conventions

7.1 Basic polymorphic constructions

Example 7.1 — Identity at unrelated instances. $\mathcal{U}_1$

```
id(A): func(x: A) -> A: x
integer: id(7)                  // 7
text:    id("seven")            // "seven"
record:  id({port: 443})        // {port: 443}
chosen:  id[int & >=0](7)       // a checked explicit instance
```

One closure meets every instance. Explicit instantiation does not construct a wrapper or remove other valid instances.

Example 7.2 — Pairing preserves an inferred common refinement. $\mathcal{U}_1$

```
pair(A: number): func(x: A, y: A) -> [A, A]: [x, y]
p: pair[int & >=1](2, 5)        // [2, 5]
q: pair(1, 2.5)                 // [1, 2.5]
```

The mixed call has the admissible instance $A = \{1, 2.5\}$. A shared type variable preserves a common refinement; it does not enforce equal primitive tags. A constant result $[0, 0]$ would satisfy the unrefined numeric result type but would fail the instance $A = \{2, 5\}$. Matching primitive categories can instead be specified by the input relation $[\mathbb{Z}, \mathbb{Z}] \cup [\mathsf{Float}, \mathsf{Float}]$.

Example 7.3 — Selecting a value preserves its subtype. $\mathcal{U}_1$

```
min(A: number): func(x: A, y: A) -> A: {
    if x <= y {out: x}
    if x > y  {out: y}
}.out

m: min[int & >=10](12, 20)      // 12
```

Either branch returns an input inhabitant. By contrast, the proposed contract `forall (A: number) func(A) -> A` does not justify adding one: $A = \{0\}$ is a counterexample. Numeric bounds permit numeric operations; result refinements require their own proof.

Example 7.4 — Two type parameters preserve asymmetric information. $\mathcal{U}_1$

```
swap(A, B): func(p: [A, B]) -> [B, A]: [p[1], p[0]]
s: swap([7, "seven"])          // ["seven", 7]
```

The two coordinates retain their distinct type variables, rather than being merged into one union.

7.2 Generic collection programs

Example 7.5 — Map with a single implementation. $\mathcal{U}_1$

```
map(A, B): func(g: func(A) -> B, xs: [...A]) -> [...B]: [
    for x in xs {g(x)}
]
inc:  func(x: int) -> int: x + 1
text: func(x: int) -> string: "\(x)"

numbers: map(inc, [1, 2, 3])    // [2, 3, 4]
strings: map(text, [1, 2, 3])   // ["1", "2", "3"]
```

The body is checked with rigid A, B. Each call creates flexible instance variables. Finite sets of admissible instantiations can contribute intersected typings, as in the CDuce polymorphic application discipline [15].

Example 7.6 — Filtering and partitioning preserve elements. $\mathcal{U}_1$

```
filter(A): func(p: func(A) -> bool, xs: [...A]) -> [...A]: [
    for x in xs if p(x) {x}
]
partition(A): func(p: func(A) -> bool, xs: [...A]) -> {
    yes: [...A]
    no:  [...A]
}: {
    yes: [for x in xs if p(x) {x}]
    no:  [for x in xs if !p(x) {x}]
}
positive: func(x: int) -> bool: x > 0
kept: filter(positive, [-2, 0, 4]) // [4]
```

The callback is pure. An effectful callback requires an explicit effect policy, including the meaning of repeated evaluation.

Example 7.7 — Flattening a generic transformation. $\mathcal{U}_1$

```
flatMap(A, B): func(g: func(A) -> [...B], xs: [...A]) -> [...B]: [
    for x in xs for y in g(x) {y}
]
duplicate(A): func(x: A) -> [A, A]: [x, x]
twice: flatMap(duplicate, [1, 2]) // [1, 1, 2, 2]
```

Example 7.8 — Mapping named record coordinates. $\mathcal{U}_1$

```
mapPoint(A, B): func(g: func(A) -> B, p: {x: A, y: A}) -> {
    x: B
    y: B
}: {
    x: g(p.x)
    y: g(p.y)
}
p: mapPoint(text, {x: 2, y: 3}) // {x: "2", y: "3"}
```

Open/closed record constraints retain their existing meaning. A separate row parameter can preserve additional fields when a row-polymorphic fragment is selected.

7.3 Higher-order composition and return values

Example 7.9 — Composition is rank-1. $\mathcal{U}_1$

```
compose(A, B, C): func(g: func(B) -> C, f: func(A) -> B) ->
    func(A) -> C:
    func(x: A) -> C: g(f(x))

pipeline: compose(text, inc)
answer: pipeline(4)           // "5"
```

The returned closure captures f and g. Its generic variables are rigid assumptions while checking the body; their run-time representations are erased.

Example 7.10 — A value-dependent closure. $\mathcal{U}_1$

```
withPrefix: func(prefix: string) -> func(string) -> string:
    func(s: string) -> string: prefix + s

warn: withPrefix("warning: ")
message: warn("retry")        // "warning: retry"
```

Two closures from the same body with different concrete prefixes have different captured environments.

Example 7.11 — A polymorphic callback. $\mathcal{S}_H$

```
useBoth: func(p: forall A func(A) -> A) -> [int, string]: [
    p(7),
    p("seven"),
]
ok: useBoth(id)              // [7, "seven"]
onlyInt: func(x: int) -> int: x
bad: useBoth(onlyInt)         // incompatible universal obligation
```

The callback itself is polymorphic. A new instantiation of the surrounding function for each use of p would not prove this contract.

Example 7.12 — A nested universal result. $\mathcal{S}_H$ *syntax; prenex normalization may apply*

```
constant(A): func(x: A) -> (forall B func(B) -> A):
    func(_) -> A: x

seven: constant(7)
a: seven("ignored")          // 7
b: seven(true)               // 7
```

The resulting closure depends on x and works uniformly for every subsequent argument type. Moving B outward is permitted only through the proven positive arrow law, with B absent from the domain.

7.4 Quantified records and independent refinement

Example 7.13 — A reusable polymorphic utility record. $\mathcal{U}_1$

```
utilities(A): {
    id:   func(x: A) -> A: x
    wrap: func(x: A) -> {value: A}: {value: x}
}
utilities: {name: "core"}

wrapped: utilities.wrap("payload") // {value: "payload"}
```

The `name` field is a constant conjunct on the same record.

Example 7.14 — A cross-file behavioral strengthening. $\mathcal{U}_1$

```
// interface.cue
wrap(A): func(A) -> {value: A}

// metadata.cue
wrap(A): func(A) -> {tag: "wrapped"}

// implementation.cue
wrap(A): func(x: A) -> {value: A, tag: "wrapped"}: {
    value: x
    tag: "wrapped"
}
```

The body is checked against both pointwise result constraints. No declaration replaces or takes precedence over another.

Example 7.15 — An intersection of domain-specific guarantees. $\mathcal{U}_1$

```
classify: func(x: int) -> ("negative" | "nonnegative"): {
    if x < 0  {out: "negative"}
    if x >= 0 {out: "nonnegative"}
}.out
classify: (func(int & <0) -> "negative") &
          (func(int & >=0) -> "nonnegative")

c: classify(-3)              // "negative"
```

An ordinary union of function values describes alternatives, whereas this intersection describes one value satisfying both implications.

Example 7.16 — Unresolved alternatives and defaults. $\mathcal{U}_1$

```
f: (func(x: int) -> int: x) |
   (func(x: int) -> int: x + 1)
a: f(3)                     // incomplete operational choice

preferred: *(func(x: int) -> int: x) |
            (func(x: int) -> int: x + 1)
b: preferred(3)             // 3, under legacy default selection
```

Constraint propagation can retain the set of alternative completions. Execution uses the ordinary concreteness/default discipline to select a callable value.

7.5 Argument forms and call packets

The five parameter forms in proposal 4484 all elaborate to packet predicates [5]. Named parameters admit a positional packet or the matching label; required and optional name-only parameters use labels; anonymous and aliased positional parameters use positions only. Packet binding precedes value constraint solving.

Example 7.17 — Positional, named, and mixed calls. $\mathcal{U}_1$

```
sum: func(a: int, b: int) -> int: a + b
p: sum(2, 3)                // 5
q: sum(a: 2, b: 3)          // 5
r: sum(2, b: 3)             // 5
```

Example 7.18 — Name-only required and optional parameters. $\mathcal{U}_1$

```
key: func(a!: int, note?: string) -> int: a
x: key(a: 4)                // 4; note is absent
z: key(a: 4, note: "audit") // 4
bad: key(4)                 // invalid packet: a is name-only
```

A body that reads an absent optional parameter must prove presence or produce a checked absence failure. Optionality is an absence alternative, not an unknown present value.

Example 7.19 — Positional-only aliases and anonymous parameters. $\mathcal{U}_1$

```
scale: func(_~value: int, _: int) -> int: value * 2
x: scale(5, 99)             // 10
bad: scale(value: 5, 99)     // invalid label/order
```

The positional alias names a local body variable without adding a public label. Renaming it preserves the packet protocol.

Example 7.20 — Hidden labels and attributes. $\mathcal{U}_1$

```
package internal
measure: func(_value: int @go(Int), unit!: string) -> int: _value
inside: measure(_value: 7, unit: "ms")
// Another package can use the positional slot, not the hidden label.
```

Hidden labels include their package identity. Attributes are retained as metadata; only a declared extension assigns them operational significance.

Example 7.21 — Errors distinguish presence from values. $\mathcal{U}_1$

```
a: sum(1, a: 1)            // duplicate binding, even with equal values
b: sum(1, 2, 3)            // surplus positional argument
c: sum(a: 1, b: 2, c: 3)    // unknown label
d: sum(1)                  // required argument missing
e: sum(_, 2)               // supplied but incomplete, not missing
```

A duplicate packet is invalid before field unification. Conjoining equal values does not erase a duplicate-binding error.

7.6 Defaults

Example 7.22 — Omission defaults and caller preferences. $\mathcal{U}_1$

```
add: func(a: int, b: int = 10) -> int: a + b
x: int | *5
omitted:  add(1)            // 11
supplied: add(1, x)         // 6
unknown:  add(1, _)         // incomplete; b was supplied
```

An omission default belongs to the closure protocol. A supplied argument bypasses it and retains its own ordinary preference information. The default expression is checked against its parameter description in the declaration environment.

Example 7.23 — A defaulted name-only argument. $\mathcal{U}_1$

```
render: func(text: string, suffix!: string = "!") -> string:
    text + suffix
x: render("hello")                // "hello!"
y: render("hello", suffix: "?")   // "hello?"
```

Defaulted required-label parameters may be omitted, while optional parameters retain absence semantics and have no omission default.

Example 7.24 — A configurable default is an explicit adapter. $\mathcal{U}_1$

```
raw: func(x: int) -> int: x
configured: func(x: int = 2) -> int: raw(x)
answer: configured()        // 2
```

The adapter supplies the protocol that accepts the empty packet. A contract mentioning that protocol requires an implementation possessing it. It does not mutate a previously supplied closure.

7.7 Open interfaces and partial application

An open signature describes a partial *protocol interface*. Elaborate its ellipsis to an existential row of further parameter declarations, retaining that witness with the function. Its callable packet domain is determined only when the required row is known. This is different from promising that every completion of the row can be omitted. $\mathcal{U}_1$ retains this row as an internal existential witness, just as it retains call-instantiation variables; explicit existential type packaging is a separate surface facility.

Example 7.25 — An open interface completed by an implementation. $\mathcal{U}_1$*, implicit protocol row*

```
T: func(a: int, ...) -> number
f: T & (func(a: int, b: int) -> int: a + b)
x: f(1, 2)                  // 3
y: f(1)                     // missing b
```

The row witness is completed by `b: int`. A closed one-argument capability admits an implementation with additional omittable parameters because its promised one-argument packets remain covered.

Example 7.26 — Chained partial application. $\mathcal{U}_1$

```
add3: func(a: int, b: int, c: int) -> int: a + b + c
f: add3(1, ...)
g: f(2, ...)
x: g(3)                     // 6
y: add3(a: 1, ...)(2, 3)    // 6
```

A partial closure stores bindings to original parameter slots. Supplied values are checked immediately; the body runs on completion. An unbound default is preserved and a bound argument suppresses that default.

Example 7.27 — A residual contract on a partial closure. $\mathcal{U}_1$

```
f: add3(1, ...)
f: func(int, int) -> int
x: f(2, 3)                  // 6
```

The residual protocol is a normal callable contract. Normalizing the original slot bindings makes partial applications comparable independently of whether arguments were supplied by label or position.

7.8 Identity, captured values, and result combination

Use a closure descriptor (o, γ, β): declaration origin o, captured free environment γ, and normalized partial bindings β. The origin survives copying; irrelevant ambient fields are absent from γ. Symbolic captures remain constraint variables.

Example 7.28 — Self-unification after copying and embedding. $\mathcal{U}_1$

```
x: {f: func(y: int) -> int: y}
a: x.f & x.f
b: x.f & {x}.f
c: x.f & (x & {g: 1}).f
r: [a(2), b(2), c(2)]       // [2, 2, 2]
```

Example 7.29 — Equal source, unequal captures. $\mathcal{U}_1$

```
d: [for v in [1, 2] {func(y: int) -> int: y + v}]
e: d[0] & d[1]             // _|_: captures differ
```

For one origin, captured-environment constraints unify structurally. Distinct known origins denote distinct intensional values. This supplies idempotence without deciding equality of mathematical functions.

Example 7.30 — Combining result descriptions explicitly. $\mathcal{U}_1$

```
left:  func(x: int) -> {a: int}: {a: x}
right: func(x: int) -> {b: int}: {b: x}
both: func(x: int) -> {a: int, b: int}: left(x) & right(x)
r: both(3)                  // {a: 3, b: 3}
```

The body is a new implementation whose result is the CUE unification of the two result descriptions. The primitive meet rule gives its result their intersection. Conflicting callbacks yield bottom on the affected calls; their consistency need not be proved when the function is defined.

7.9 Validators and iteration

Example 7.31 — A predicate used as an ordinary constraint. $\mathcal{U}_1$

```
positive: func(x: int) -> bool: x > 0
#Positive: {value: int, _valid: true & positive(value)}
n: (#Positive & {value: 7}).value   // 7
bad: #Positive & {value: -1}       // _|_
```

A validator is a predicate on an existential subject. It can be expressed with existing struct machinery once predicate functions are available.

Example 7.32 — Portable foreign contract. $\mathcal{U}_1$ *with checked effects*

```
externF: extern func(int) -> int !bridge
// A native-range failure is a bridge outcome.
// A fractional input is outside the logical domain.
```

The bridge checks logical kind, representation, conversion, and returned values as specified by the source proposal [6]. The effect annotation records those failures; it introduces no additional conversion stage. Tool-level effects execute in an explicit sequencing context, while constraint unification operates on their descriptions.

Example 7.33 — Nested calls remain ordinary evaluation. $\mathcal{U}_1$

```
twice: func(n: int) -> int: n + n
value: twice(twice(twice(2))) // 16
```

The initial execution profile preserves the host cycle discipline. A structurally recursive extension can admit calls justified by a decreasing finite measure. This choice is independent of quantifier rank.

Example 7.34 — A structurally recursive fold. $\mathcal{U}_1$ *with structural recursion*

```
fold(A, B): func(step: func(B, A) -> B, seed: B, xs: [...A]) -> B: {
    if len(xs) == 0 {out: seed}
    if len(xs) > 0 {out: fold(step, step(seed, xs[0]), xs[1:])}
}.out
plus: func(x: int, y: int) -> int: x + y
sum: fold(plus, 0, [1, 2, 3]) // 6
```

Each recursive call reduces list length. The termination proof is independent of polymorphism; a nonrecursive profile may supply the same contract as a trusted primitive.

Chapter 8

Dependent descriptions and functions

8.1 Dependent contexts

A dependent context extends Γ by a value x satisfying a description A in that context:

$$\mathrm{Env}(\Gamma, x : A) = \{(\rho, x) \mid \rho \in \mathrm{Env}(\Gamma),\ x \in [\![A]\!]_\rho\}.$$

The existential and universal adjoints already apply to its projection. Dependence therefore changes the fibers over which quantification ranges, rather than introducing another interpretation of refinement.

A dependent function contract uses its actual argument in the result predicate:

$$\Pi_{x:A} B(x) = \{f \in F \mid \forall x \in A.\ H_{B(x)}(f, x)\}.$$

A dependent pair retains the witness:

$$\Sigma_{x:A} B(x) = \{(x, y) \mid x \in A,\ y \in B(x)\}.$$

Logical $\forall x\, B(x)$ constrains *one subject* at every x; a dependent function instead relates inputs to results. Logical $\exists x\, B(x)$ projects away x; a dependent pair retains it. The distinction is determined by the subject and the application or pairing relation.

Design rule. Within a dependent function signature, parameters enter scope from left to right. A result constraint can refer to all parameters. Default expressions can depend only on parameters whose binding is already established. The dependency graph determines evaluation order; it is independent of the source order of unrelated record declarations.

Example 8.1 — A directly dependent result. $\mathcal{U}_1{+}\mathcal{D}$

```
successor: func(x: int) -> (x + 1): x + 1

// Equivalent pointwise specification:
successor(n in int): func(n) -> (n + 1)
```

The second form quantifies over input singletons. The first uses a dependent result predicate on the actual argument.

Example 8.2 — A dependent parameter constraint. $\mathcal{U}_1{+}\mathcal{D}$

```
interval: func(lo: int, hi: int & >=lo) -> {
    low: lo
    high: hi
}: {low: lo, high: hi}

ok: interval(3, 7)          // {low: 3, high: 7}
bad: interval(7, 3)         // domain contradiction
```

The admitted domain is a relation on two arguments. It is not the Cartesian product of their marginal types.

8.2 Indexed finite sequences

The following aliases use the list validators supplied by CUE [4]:

```
import "list"

let Nat = int & >=0
Vec(A, n in Nat) = [...A] &
    list.MinItems(n) & list.MaxItems(n)
Fin(n in Nat) = int & >=0 & <n
```

Their reference semantics is

$$[\![\mathrm{Vec}(A,n)]\!] = \{xs \in \mathsf{List}(A) \mid |xs| = n\}, \qquad [\![\mathrm{Fin}(n)]\!] = \{0,\dots,n-1\}.$$

A symbolic index remains a constraint variable. The checker uses the length predicate and its lemmas; concrete validator execution is one implementation strategy for ground indices.

Example 8.3 — Length-preserving map. $\mathcal{U}_1{+}\mathcal{D}$

```
map(A, B, n in Nat): func(
    g: func(A) -> B,
    xs: Vec(A, n),
) -> Vec(B, n): [for x in xs {g(x)}]
```

The universal n records a property of the input, rather than supplying a runtime integer. Its value is available as `len(xs)` when evaluation requires it.

Example 8.4 — Safe indexing, including the zero-length case. $\mathcal{U}_1{+}\mathcal{D}$

```
index(A, n in Nat): func(xs: Vec(A, n), i: Fin(n)) -> A: xs[i]

x: index(["a", "b"], 1)    // "b"
bad: index([], 0)           // 0 is outside Fin(0)
```

For $n = 0$, the packet domain is empty. The generic implementation remains valid; there is no admitted out-of-range call.

Example 8.5 — Head with an explicit nonempty precondition. $\mathcal{U}_1{+}\mathcal{D}$

```
head(A, n in int & >=1): func(xs: Vec(A, n)) -> A: xs[0]
first: head([4, 5])         // 4
bad: head([])               // no admissible instance
```

Example 8.6 — Appending adds lengths. $\mathcal{U}_1{+}\mathcal{D}$

```
append(A, n in Nat, m in Nat): func(
    xs: Vec(A, n), ys: Vec(A, m),
) -> Vec(A, n + m): [for x in xs {x}, for y in ys {y}]

v: append([1, 2], [3])      // [1, 2, 3]
```

The proof combines the cardinalities of two concatenated finite comprehensions.

Example 8.7 — Zip rejects unequal lengths. $\mathcal{U}_1{+}\mathcal{D}$

```
zip(A, B, n in Nat): func(xs: Vec(A, n), ys: Vec(B, n)) ->
    Vec([A, B], n): [for i, x in xs {[x, ys[i]]}]

z: zip([1, 2], ["a", "b"]) // [[1, "a"], [2, "b"]]
bad: zip([1], ["a", "b"])   // inconsistent length witness
```

The same existential instance of n must satisfy both input constraints at a call. Separate instances per argument would lose this invariant.

Example 8.8 — Split at a bounded position. $\mathcal{U}_1+\mathcal{D}$

```
split(A, n in Nat, k in int & >=0 & <=n): func(
    xs: Vec(A, n), cut: k,
) -> {left: Vec(A, k), right: Vec(A, n - k)}: {
    left: xs[:cut]
    right: xs[cut:]
}
```

The runtime parameter `cut` reifies the index used by slicing. The endpoints $k = 0$ and $k = n$ are admitted.

Example 8.9 — Replication reifies its count. $\mathcal{U}_1+\mathcal{D}$

```
replicate(A, n in Nat): func(x: A, count: n) -> Vec(A, n): [
    for i in list.Range(0, count, 1) {x}
]
r: replicate("x", 3)      // ["x", "x", "x"]
```

An erased type or value index cannot be inspected by a body merely because it appears in a contract. Here the actual count parameter supplies the needed runtime information. A checked library bridge can carry its declared allocation or representation effects; the finite reference range operation is total.

Example 8.10 — A dependent result with a hidden length. $\mathcal{S}_H+\mathcal{D}$

```
filterBounded(A, n in Nat): func(
    p: func(A) -> bool, xs: Vec(A, n),
) -> (exists (k in int & >=0 & <=n) Vec(A, k)):
    [for x in xs if p(x) {x}]
```

The existential witness can depend on the input and predicate. A $\mathcal{U}_1+\mathcal{D}$ surface can express the same projected result using `[...A] & list.MaxItems(n)`. A witness-bearing version returns a record `{length: k, values: Vec(A,k)}`.

8.3 Matrix dimensions, including empty matrices

Retain dimensions as data when empty collections do not determine them:

```
Matrix(A, r in Nat, c in Nat) = {
    rows: r
    cols: c
    data: Vec(Vec(A, c), r)
}
```

A zero-row matrix still records its column count. This makes transposition and composition defined at all admitted dimensions.

Example 8.11 — Transpose with a precise shape. $\mathcal{U}_1+\mathcal{D}$

```
transpose(A, r in Nat, c in Nat): func(m: Matrix(A, r, c)) ->
    Matrix(A, c, r): {
        rows: m.cols
        cols: m.rows
        data: [
            for j in list.Range(0, m.cols, 1) {
                [for row in m.data {row[j]}]
            }
        ]
    }
```

Every j is below the length c of every row. The outer length is c and the inner length is r, including either zero dimension.

Example 8.12 — Dot product with matched dimensions. $\mathcal{U}_1+\mathcal{D}$*; finite fold primitive*

```
addNumber: func(x: number, y: number) -> number: x + y

dot(n in Nat): func(xs: Vec(number, n), ys: Vec(number, n)) ->
    number: fold(addNumber, 0, [for i, x in xs {x * ys[i]}])
```

Products and sums return `number`; arbitrary input refinements are not assumed closed under arithmetic.

Example 8.13 — Dimension-safe matrix multiplication. $\mathcal{U}_1+\mathcal{D}$

```
matmul(r in Nat, k in Nat, c in Nat): func(
    x: Matrix(number, r, k), y: Matrix(number, k, c),
) -> Matrix(number, r, c): {
    rows: x.rows
    cols: y.cols
    data: [
        for row in x.data {
            [for j in list.Range(0, y.cols, 1) {
                dot(row, [for yrow in y.data {yrow[j]}])
            }]
        }
    ]
```

A single instance of k links the two inputs and justifies each dot product. The output carries the surviving dimensions r, c.

8.4 Protocol descriptions and refinement predicates

Example 8.14 — A request-indexed response. $\mathcal{U}_1+\mathcal{D}$ *with a checked service*

```
let Request = {kind: "lookup", key: string} |
              {kind: "size", items: [..._]}
Reply(q in Request) = {
    if q.kind == "lookup" {result: string}
    if q.kind == "size"   {result: int & >=0}
}
request: extern func(q: Request) -> Reply(q) !service
```

The result predicate is evaluated in the request's environment. Replacing it by an independent union of responses would admit mismatched request/response pairs.

Example 8.15 — A relation between input and output records. $\mathcal{U}_1+\mathcal{D}$

```
annotate(A): func(x: A, tag: string) -> {
    value: x
    label: tag
}: {value: x, label: tag}
```

The field labels differ from the parameter names. The result predicate states equality with the corresponding argument cells and retains the input/output correlation through subsequent record refinement.

Example 8.16 — Clamping to an argument-dependent interval. $\mathcal{U}_1+\mathcal{D}$

```
clamp: func(lo: int, hi: int & >=lo, x: int) -> (
    int & >=lo & <=hi
): {
    if x < lo {out: lo}
    if x > hi {out: hi}
    if x >= lo && x <= hi {out: x}
}.out

bounded: clamp(0, 10, 14) // 10
```

The three branches partition the input domain. The dependency $hi \geq lo$ justifies each branch's interval postcondition. The result is established by linear arithmetic and branch refinement.

8.5 Proof fragments and erasure

Dependent specifications can be verified by a hierarchy of procedures: structural length reasoning, quantifier-free linear integer arithmetic, finite disjunction splitting, and explicit proof certificates for stronger predicates. Refinement-type inference provides a precedent for combining type structure with restricted automated implication checking [22].

The logical reference model admits well-formed dependent predicates. Every implementation-to-contract implication is proved before certification. A difficult first-order predicate can instead be tested by an explicit source assertion such as `x & >0`. The assertion has a local typing rule that establishes its predicate on successful results; executing it can fail or remain pending. Section 11.6 defines this boundary. Predicates on functions, abstract types, or infinite universal ranges require a static proof or an explicitly justified source operation.

All function-type annotations, type binders, instantiation selections, and proof arguments erase. Executable assertions remain because they are terms. Value indices used computationally remain ordinary arguments or fields, as in replication and matrices. Theorem 11.4 gives erasure for both inline and separately contributed annotations. Explicit refinement systems provide a related separation of proof evidence from executable code [30].

Chapter 9

Existentials and modular reuse

9.1 A shared hidden representation

An existential record describes several fields using one representation witness:

```
#Counter: exists State {
    zero: State
    next: func(State) -> State
    read: func(State) -> (int & >=0)
}
```

Its meaning is $\bigcup_{S \in \mathcal{U}} [\![\text{Counter}(S)]\!]$. One S must make all three fields satisfy the interface. Replacing it with separate existential witnesses for the three fields loses the relationships that make composition type-correct.

This is the type-theoretic basis of abstract data types and modules identified by Mitchell and Plotkin [18]. In CUE, an additional distinction matters: existential projection is a logical operation, while *information hiding* is a restriction on the observations available at an interface.

9.2 Sealing and opening

A *seal* turns an implementation of an existential interface into an opaque public view. For an unbounded representation sort S, a private representation A is exposed through a fresh abstract copy

$$\widehat{A}_\kappa = \{(\kappa, a) \mid a \in A\},$$

where κ is a seal identity. Exported operations wrap and unwrap that copy through authorized adapters. The public existential witness is $\widehat{A}_\kappa$. Thus the set interpretation of the interface remains an existential union.

Public operations are exposed through opaque capability handles associated with the seal and the interface's declared export/alias graph. Their private code origins and captures are not observable through that view. Copying a sealed module preserves its seal and exported aliasing; explicitly constructing a fresh sealed module introduces a fresh seal.

Sealing and opening have the forms

```
seal Interface with (HiddenType = Representation) {
    // implementation fields, checked with the private representation
}

open moduleValue as (AbstractType, View) {
    // AbstractType is rigid; View has the opened interface.
}
```

Opening introduces a local rigid type and preserves sharing across the view's field projections. The type cannot escape except under a corresponding existential package. $\mathcal{S}_1$ uses this operation at module boundaries; $\mathcal{S}_H$ admits it inside functions over first-class packages.

A transparent bounded existential still exposes every operation justified by its bound. A fully opaque initial sealing profile uses unbounded representation sorts and explicit exported operations. A bound such as `State: number` would expose numeric structure and entails a stronger observation-preservation condition on any representation change.

9.3 Two implementations and one client

Example 9.1 — An integer representation. $\mathcal{S}_1{+}\mathcal{A}$

```
counterV1: seal #Counter with (State = int & >=0) {
    zero: 0
    next: func(x: int & >=0) -> (int & >=0): x + 1
    read: func(x: int & >=0) -> (int & >=0): x
}
```

Example 9.2 — A record representation. $\mathcal{S}_1{+}\mathcal{A}$

```
let CounterRep = close({count: int & >=0})
counterV2: seal #Counter with (State = CounterRep) {
    zero: {count: 0}
    next: func(s: CounterRep) -> CounterRep:
        {count: s.count + 1}
    read: func(s: CounterRep) -> (int & >=0): s.count
}
```

The abstract carrier in each public view has no accessible `count` field or numeric representation. Its structure is the interface's structure.

Example 9.3 — A client checked once. $\mathcal{S}_1{+}\mathcal{A}$ *with a rank-1 utility*

```
useCounter(S): func(c: {
    zero: S
    next: func(S) -> S
    read: func(S) -> (int & >=0)
}) -> (int & >=0): c.read(c.next(c.next(c.zero)))

v1: (open counterV1 as (S, C) {out: useCounter[S](C)}).out
v2: (open counterV2 as (S, C) {out: useCounter[S](C)}).out
// Both observations are 2.
```

The client uses only the abstract operations. Its generic declaration is rank-1. The hidden type is opened at the module boundary and then supplied as a rigid instance.

Example 9.4 — A module law strengthened by universals. $\mathcal{S}_1{+}\mathcal{D}{+}\mathcal{A}$

```
#CounterLawful: exists State {
    zero: State
    next: func(State) -> State
    read: func(State) -> (int & >=0)
    _zeroLaw: true & (read(zero) == 0)
    forall (s in State) {
        _stepLaw: true & (read(next(s)) == read(s) + 1)
    }
}
```

The law is a proof obligation on the module subject, not an instruction to enumerate all states. Both implementations discharge it by substitution into their bodies. The existential representation scope encloses the universal state-law scope.

9.4 Unification is an observation

Opaque values can participate in CUE unification. In the basic sealed model,

$$(\kappa, a) \,\&\, (\kappa, b) = \begin{cases} (\kappa, a), & a = b, \\ \bot, & a \neq b. \end{cases}$$

Consequently, a representation-independence theorem must preserve equality and conflict, not merely the outputs of named methods.

Definition 9.1 (Equality-respecting representation simulation)**.** *For two representations A_1, A_2, a simulation is a bijection $r : A_1 \to A_2$ on the entire carriers admitted by the public interface, together with corresponding exported capability handles, such that constants correspond, every public operation commutes with r on successful results, and failure, divergence, and public effects correspond as well. The carriers include the abstract states over which a client may quantify, as well as those reached by executing operations.*

For the counters, $r(n) = \{count : n\}$ is such a bijection. It preserves zero, successor, reading, and equality. The abstract public seal identities are related consistently across the two executions.

Extend r componentwise to data constructors and to sets of completions by direct image. A bijection preserves intersections, unions, emptiness, and singletons. Its transport of abstract type assignments induces bijections on the corresponding admissible predicate families.

Theorem 9.2 (Representation-independent replacement)**.** *Assume an equality-respecting simulation, scope-preserving sealing and opening, and admissible type families and public operational inhabitants closed under the induced transport. Let the client's observations be confined to the declared public interface, ordinary data, unification, and the admitted effects. Replacing one sealed implementation by the related implementation preserves every client data observation and its success/conflict outcome.*

Proof. Interpret the client in the two related public environments. Constants and public operations preserve the relation by hypothesis. Reindexing and data constructors preserve it componentwise. Direct image along a bijection commutes with unions and intersections. Quantifiers range over corresponding assignments, so their unions and intersections commute with transport. Application preserves the relation by the operation simulation. Sealing preserves the public alias structure and hides implementation descriptors. Induction over the elaborated client derivation therefore relates its resulting completion predicates. At ordinary exported data, the relation is equality; emptiness also agrees. □

The abstraction discipline is related to the classical logical-relations proof of representation independence [19, 10]. The explicit equality condition is required by CUE's observable meet operation.

A many-to-one abstraction relation requires additional treatment. If multiple private cache states represent the same logical state, their raw equality can be observable through unification. An interface may instead expose a certified quotient carrier and an equality congruence. Its sealing adapter then presents logical equivalence classes, and the theorem applies to the resulting public carrier. The implementation must realize that equality soundly.

9.5 The scope of the evolution guarantee

A partial function interface alone does not ensure preservation of successful calls across replacement. The outcome-preserving simulation in theorem 9.2 is the additional condition: replacing a successful operation by one that always fails has the same partial result type but violates that simulation. Required law fields are also preserved as observations.

The theorem yields a checkable class of representation changes for which all admissible clients continue to work. Its hypotheses include behavior and observable equality. A signature that merely types `next` and `read` does not assert that `read(next(s)) = read(s)+1`; that equation is a separate interface law or simulation obligation.

For one-way evolution, an observation-preserving simulation can relate the old interface to a designated old-compatible view of the new implementation. New operations and states may be exposed through an extended view. If additional states are exposed through the old abstract type itself, preservation also requires every old universally quantified obligation to hold on those states; agreement merely on previously executed states is insufficient. Effect guarantees and public serialization are part of the same simulation.

Here a *forward-compatible client* is a client whose old interface contract continues to hold when a later, related implementation is substituted. This is the usual backward-compatible replacement direction for the library. The quantified interface and simulation state the precise class of permitted future changes.

Ordinary exposed structural schemas already support structural refinement and interface checks. Sealed existential witnesses add a generic *representation-independent* replacement guarantee. A private field alone does not abstract the representation of a public value whose numeric, record, or list structure remains observable.

9.6 First-class packages and existential placement

Example 9.5 — Heterogeneous values with their own operations. $\mathcal{S}_H{+}\mathcal{A}$

```
#Showable: exists A {
    value: A
    show: func(A) -> string
}
p: seal #Showable with (A = int) {
    value: 7
    show: func(x: int) -> string: "\(x)"
}
q: seal #Showable with (A = string) {
    value: "hello"
    show: func(x: string) -> string: x
}
items: [p, q]
```

Each package has its own type witness and seal. A list of packages differs from one package containing a homogeneous list.

Example 9.6 — Eliminating a first-class package. $\mathcal{S}_H{+}\mathcal{A}$

```
describe: func(p: #Showable) -> string:
    (open p as (A, P) {out: P.show(P.value)}).out
text: map(describe, items)     // ["7", "hello"]
```

At a sealed interface, projections are mediated by the opened abstract view. The checker does not independently instantiate the hidden type for `show` and `value`.

Example 9.7 — Sharing two values under one witness. $\mathcal{S}_H{+}\mathcal{A}$

```
#ComparablePair: exists A {
    left: A
    right: A
    equal: func(A, A) -> bool
}
compare: func(p: #ComparablePair) -> bool:
    (open p as (A, P) {out: P.equal(P.left, P.right)}).out
```

Separate existential packages for the two operands would require an explicit sharing certificate before one comparator could consume both.

Example 9.8 — An abstract codec interface. $\mathcal{S}_1{+}\mathcal{A}$*; checked effects*

```
#CodecModule: exists Value {
    encode: func(Value) -> bytes
    decode: func(bytes) -> Value !decode
}
```

A round-trip law requires decoding an encoded value to return the corresponding abstract value. Persisted wire compatibility additionally constrains the chosen byte representation; hiding an in-memory representation does not silently version an external protocol.

9.7 Mixed prefixes and generic modules

The order of module binders expresses representation dependence:

```
module(A): {
    exists State
    empty: State
    push: func(A, State) -> State
}
```

The logical order is $\forall A\, \exists State$. It allows the hidden semantic state type to depend on A while constraining one uniform module subject. A standard witness is $State = \mathsf{List}(A)$, with empty list and a generic cons operation. It differs from $\exists State\, \forall A$, which chooses one semantic state type before the element type varies.

When different *runtime* modules are required for different arguments, an ordinary module-producing function supplies the module subject. Quantifiers specify its interface; they do not themselves allocate a record at each type assignment. This keeps generativity, runtime allocation, and logical dependence separate and explicit.

Chapter 10

Checking, execution, and formal obligations

10.1 Typing judgments and operational subjects

Use rigid type variables, term variables, and scoped existential witnesses in Γ. Computation typing has synthesis and checking forms

$$\Gamma \vdash e \Rightarrow T, \qquad \Gamma \vdash e \Leftarrow T.$$

Their conclusion constrains successful results. Value membership is separately written $\Gamma \vdash_v v : T$. Subtyping certificates establish $\Gamma \vdash_{\mathcal{K}} T \leq U$ and are sound for semantic inclusion. A compiler may fail to establish such a certificate without proving the denotation empty.

Universal introduction checks the same erased computation under a fresh rigid type variable. Universal elimination substitutes any well-sorted source type permitted by the formation profile, including a polytype in $\mathcal{S}_H$. Existential elimination opens one rigid witness shared by all projections and checks its escape condition. Scope levels record dependencies, not universes.

Bidirectional typing separates annotation checking from inference [28]. Explicit type abstraction and instantiation provide a route to checking every annotated System F derivation. Inference of missing impredicative arguments is an optional front-end service, never a condition on the decidability of checking the supplied elaboration.

10.2 Partial correctness of typed computation

A lambda is a value even when its body may fail. The lambda rule checks the body under its parameter types. Application checks its function and argument types before using the result type. It never assumes success of the body.

Theorem 10.1 (Partial-correctness soundness)**.** *Assume sound primitive rules, sound subtyping certificates, scope-correct elaboration, and the explicit-assertion rules of chapter 11. If $\Gamma \vdash e \Leftarrow T$, then in each environment satisfying Γ, every successful complete result of the elaborated computation lies in $[\![T]\!]$. If a resulting closure is certified at $A \Rightarrow B$, a successful call with an admitted A-packet returns only a B-result. Failure and divergence are permitted; a computation cannot produce a successful value by subsuming failure to another type.*

Proof. Induct on the derivation. Constants and structural values use membership; subsumption uses inclusion. The lambda case establishes the conditional universal in equation (4.1), and application instantiates it. Meet intersects successful completion predicates. Universal introduction varies the rigid valuation without changing erased code; elimination uses proposition 2.3. Existential rules preserve their shared witness. Source assertions forward only validated successes and otherwise fail or remain pending. Strict evaluation premises and retained demand dependencies distinguish failed computations from returned values. □

The theorem imposes no termination premise. A separately selected execution profile may reject recursive code or permit a fixed-point operator with its standard partial type. A derivation from an inconsistent data hypothesis is not an implementation certificate: source checking uses verified variable types, not arbitrary unresolved propositions as axioms for ex falso.

10.3 Symbolic evaluation and delayed structure

For a symbolic expression e, retain the successful-completion predicate $\mathrm{CompObs}(e)$ and its demand and failure dependencies. The exact normalization target is

$$[\![N(e)]\!] = \mathrm{CompObs}(e).$$

Abstract shape and bound information may guide propagation while the exact residual graph remains present. A candidate concrete result with a pending condition is an observation under that condition, not an unconditional export.

CUE records and arguments can contain suspended computations. A required field whose successful-completion predicate is proved empty makes the record's completion predicate empty. The normalizer propagates that refutation through its required enclosing fields; a suspended implementation must not indefinitely hide an established contradiction. Optional impossible fields instead constrain presence, according to ordinary CUE field rules.

Static expression checking nevertheless verifies every subexpression from its own justified interface. A contradiction elsewhere in a record does not excuse an invalid field selection or produce a certificate for an unrelated expression. Within a function, a proof that the body fails under certain arguments is scoped to those arguments. It does not turn the returned closure into a failed closure-construction computation. The value/computation distinction for non-strict semantic subtyping is relevant here [31].

The reference System F sublanguage uses strict application to value arguments. A demand-driven evaluator can postpone computation, but commits a completed call only when its required argument and result dependencies have been validated. It must not use the empty-domain arrow identity to discard an unvalidated argument and produce an arbitrarily typed result. Memoization preserves these dependencies as well as code identity and captured bindings.

10.4 Static certificates and executable assertions

Every implementation-to-interface obligation has two destinations: a checked certificate or a blocking diagnostic. This applies to structural, kind, literal, polymorphic, callable-protocol, and dependent-result requirements alike. An unresolved implication does not become an inserted runtime check.

An explicitly written assertion has a separately checked executable meaning. Its successful-result predicate contributes to the body's inferred type, while its execution can remain pending on data. The pending operation is source code, not an unproved type obligation. This distinction permits cheap certification of `x & >0` without proving that every possible `x` is positive.

An unspecified implementation may remain an ordinary data hypothesis after its explicit interface passes relevant consistency and eager refutation. The interface check does not wait for a body or an observing call. A client can be checked under that interface; linking verifies the actual implementation against that interface. A completed executable artifact has neither unchecked imports nor outstanding annotation obligations. The stages and minimum judgments are fixed in chapter 11.

Chapter 11

Declarative checking and explicit assertions

11.1 The checking contract

The checking profile $\mathcal{K}$ fixes a mandatory static fragment independently of the formation profiles $\mathcal{U}_1$, $\mathcal{S}_1$, and $\mathcal{S}_H$, and the extensions $\mathcal{D}$ and $\mathcal{A}$. Every annotated implementation receives a complete certificate before executable use. Every explicit interface first passes the relevant-consistency check of section 11.5, including an interface with no implementation. Body refinements may follow from symbolic reasoning or from the successful-result rules of assertions already in its body.

Elaboration produces

$$(e^\circ,\ \pi,\ R),$$

where e° is the source program with its type annotations erased, π certifies its interface and the source relevance check, and R retains the exact data constraints and execution dependencies of its explicit operations. No executable node in R is created solely to satisfy an annotation. The observable outcomes of checking are:

1. *Accepted*: explicit interfaces pass relevant consistency and all typing and interface implications have certificates; source assertions may still await their data.
2. *Blocked*: an invalid relevant observation, a missing annotation or proof, an unsupported operation, an unproved implication, or exhausted static resources.
3. *Refuted*: a checked empty value or completion predicate, propagated at its scope.

A refuted required data subject is bottom. A refuted function-body path becomes a failing computation at that path, rather than an unrelated global contradiction. A blocking diagnostic does not assert semantic emptiness.

A module interface may remain open until linking. Refining that interface triggers the same accumulated-conjunction check; it does not attach executable checks to a cached closure. Completing missing declarations may resolve a blocking diagnostic, but cannot remove a proved contradiction.

11.2 The mandatory type fragment

The strict grammar is finite, alias-expanded type syntax:

$$\begin{aligned} T, U ::= {} & \bot \mid \top \mid \alpha \mid k \mid c \mid T \wedge U \mid T \vee U \mid T \Rightarrow U \\ & \mid \{\ell_i^{m_i} : T_i\}_{i \in I} \mid [T_1, \ldots, T_n] \mid \mathsf{List}(T) \mid \forall \alpha\, T \mid \exists \alpha\, T. \end{aligned}$$

Here k ranges over CUE kinds and c over literals. Field mode m_i records requiredness or optionality, together with an explicit absence constraint when present. The permitted quantifier positions are those of the selected formation profile. Callable packets carry their statically declared arity, labels, requiredness, defaults, and bound slots. Recursive aliases are outside this finite kernel unless equipped with a separately specified contractive regular-type procedure.

The fragment includes all of the following, not just record subtyping:

- kind membership and the kind lattice, including $\mathbb{Z} \leq \mathsf{Number}$, $\mathsf{Float} \leq \mathsf{Number}$, and $\mathsf{Number} = \mathbb{Z} \cup \mathsf{Float}$ and $\mathsf{Bool} = \{\mathsf{false}, \mathsf{true}\}$;
- literal equality, literal membership in kinds, and literal inclusion in finite unions and intersections;
- record width, depth, presence, and required-field checks; fixed list lengths and component types; homogeneous lists and their element types;
- contravariant arrows, finite union/intersection reasoning, and scoped universal and existential introduction and elimination.

The kind table distinguishes integers from decimal floating-point values, strings from bytes, and data kinds from closures. Literal comparison and arithmetic preserve CUE's specified numeric representation rules [1].

Consequently both

$$\{a : 1, b : \mathsf{true}\} \le \{a : 1\}, \qquad (1 \vee 2) \le \mathbb{Z}$$

are mandatory, although only the first is structural subtyping in the record sense. A kernel cannot postpone either judgment to a runtime cast.

11.2.1 Records are closed for typing

The parameter's declared field inventory is closed for static reasoning: unmentioned fields cannot be invented to justify selection or application. Width subtyping remains available. The record type $\{a : 1\}$ guarantees the field a and may describe a value also having b; only a is accessible through that static interface. This is distinct from `close({a: 1})`, which also forbids extra fields in the data predicate.

Interpreting every bare type as an exact-label set would contradict the required width relation, so the two notions of closure must remain separate. Extra fields on an argument are permitted by width subtyping unless explicit data closedness forbids them. Missing required fields are blocked, never guessed from an open ambient schema.

Example 11.1 — Closed static field inventory. $\mathcal{U}_1{+}\mathcal{K}$

```
get: func(r: {a: int}) -> int: r.a
ok: get({a: 1, b: true})        // 1: width subtyping

bad: func(r: {a: int}) -> int: r.b
// blocked at definition: b is absent from the parameter interface

needsB: func(r: {a: int, b: bool}) -> int: r.a
badCall: needsB({a: 1})         // blocked: required field b is missing
```

An explicit occurrence test can refine an optional field to present. A finite union of record shapes is checked branchwise. A general dynamic-label map requires its own declared map interface; it is not an excuse to residualize a missing field in a finite record type.

11.3 Strict typing rules

The internal language records lambda annotations, type abstractions, type applications, and the branch choices used by union/intersection derivations. Source inference may insert them. The kernel checks the resulting proof graph without searching for arbitrary types.

Representative mandatory rules are

$$\frac{\Gamma, x : S \vdash e \Leftarrow T}{\Gamma \vdash \lambda(x : S).e \Leftarrow S \Rightarrow T} \qquad \frac{\Gamma \vdash f \Rightarrow S \Rightarrow T \quad \Gamma \vdash x \Leftarrow S}{\Gamma \vdash f(x) \Rightarrow T}.$$

These are computation judgments: success has the displayed type. They do not assert that evaluating the computation produces a value. The rules for explicit polymorphism are

$$\frac{\Gamma, \alpha : \mathsf{Type} \vdash e \Leftarrow T \quad \alpha \notin \mathrm{FV}(\Gamma, e_{\mathrm{run}})}{\Gamma \vdash e \Leftarrow \forall \alpha\, T} \qquad \frac{\Gamma \vdash e \Rightarrow \forall \alpha\, T \quad S : \mathsf{Type}}{\Gamma \vdash e[S] \Rightarrow T[S/\alpha]}.$$

S may contain quantifiers in $\mathcal{S}_H$. Checking the substitution is mandatory; inferring S is not. The corresponding existential rules choose or open one shared witness with its usual escape condition.

Intersection introduction checks the same expression at both types. Union introduction records a proved injection; union elimination checks every possible incoming branch. Subsumption requires a checked inclusion derivation. A supplied literal synthesizes its singleton, and a supplied record synthesizes its present fields. These precise descriptors are not silently widened before checking a singleton or required-field obligation.

For unification expressions, the mandatory primitive rule is

$$\frac{\Gamma \vdash e \Rightarrow S \qquad \Gamma \vdash d \Rightarrow T}{\Gamma \vdash e \,\&\, d \Rightarrow S \wedge T}. \tag{11.1}$$

The result can fail. Both operands are checked; the possibility of a conflict cannot conceal an ill-typed subexpression. A known local conflict is diagnosed by the eager rules below. An explicit failure computation has type $\mathsf{Comp}(\bot)$ and can occur in a function body or a deliberate failing branch. It supplies no successful witness and permits no arbitrary cast of another expression.

Example 11.2 — Annotations assert properties of the body. $\mathcal{U}_1+\mathcal{K}$

```
id(A): func(x: A) -> A: x          // accepted
wrong(A): func(x: A) -> A: 0       // blocked: 0 is not arbitrary A
asText: func(x: int) -> string: x  // blocked: int does not imply string
asTwo: func(x: int) -> 2: 1        // refuted singleton obligation
stop(A): func(x: A) -> A: _|_      // explicit failure; no successful return
```

Every result annotation generates the same implication judgment. The body is checked before result subsumption; failure of that implication blocks the annotation. A source meet uses equation (11.1); no meet is added by checking. The explicit failure term has no successful result and is checked by its own primitive rule, after all enclosing structural obligations have been verified.

11.3.1 A finite subtyping proof calculus

Every kernel verifies reflexivity, transitivity with an explicit intermediate type, bottom/top, the finite kind table, and literal membership. It verifies record width/depth and presence rules, covariant products/lists, and contravariant arrows. It also verifies the lattice rules

$$S \wedge T \le S, \quad S \le S \vee T, \quad \frac{R \le S \quad R \le T}{R \le S \wedge T}, \quad \frac{S \le R \quad T \le R}{S \vee T \le R},$$

finite distributivity, and the arrow laws in proposition 4.1. Quantified inclusions carry explicit instantiations, openings, or the pointwise rules of chapter 3. Bounded universal comparison uses the same bound under a rigid variable, or a separately supplied proof of the required guarded inclusion; unrestricted bounded-subtyping proof search is not a kernel rule.

This defines a proof-checkable extension of explicitly typed System F with structural subtyping, unions, intersections, literals, and CUE kinds. Every finite derivation whose explicit interfaces pass relevant consistency is accepted when its proof graph fits the published resource bound. Inferred and intermediate types may contain empty successful-result predicates; their use in a derivation does not turn them into source interface declarations. It is not a completeness claim for every inclusion between impredicative denotations. A smaller syntactic inclusion calculus can be sound even when semantic inclusion has additional valid statements.

A practical front end memoizes type-pair obligations, normalizes finite record fields and kind/literal unions, splits source unions and target intersections, and tries the finitely recorded target-union/source-intersection alternatives. Structural arrow checks recurse on smaller component types. Quantified elimination uses explicit source instances or a finite candidate list; it never searches an unbounded space of polytypes. Optional finite Boolean normalization and the displayed distributive laws can derive more inclusions within the same budget. A supplied proof can resolve a goal the search did not find. Explicitly annotated higher-rank checking with union/intersection types and impredicative applications has a closely related mechanized precedent [29].

11.4 Required eager refutation

Before issuing a certificate, run a shared eager-refutation service over the available concrete and abstract descriptors. The service is also used by ordinary CUE normalization. A conforming implementation publishes its finite transfer rules and the conformance corpus for its chosen base-language version. Type checking must preserve every refutation that this service produces; it cannot downgrade one to a dynamic residual.

The mandatory minimum includes:

1. incompatible known kinds and unequal atomic literals, including numeric kind distinctions and finite literal-union exclusion;
2. empty intersections of explicit constant numeric intervals, with open endpoints and integer rounding handled correctly; concrete pattern checks;
3. known conflicting required fields, forbidden fields in explicitly closed data, impossible field-presence combinations, and incompatible fixed list lengths or components;
4. exact evaluation of enabled ground primitives and comparisons, invalid known indexes, malformed call packets, duplicate labels, and missing arguments;
5. rejection of a finite alternative only after that alternative is refuted, and rejection of their union only after all alternatives are refuted.

Optional impossible fields describe absence and are not automatically an inconsistent record. Required impossible fields propagate bottom. Branch and quantifier assumptions remain attached to refutations; a counterexample under an impossible guard proves nothing about a live branch. A contradiction in an uncalled function body identifies a failing body path. It does not remove the closure value or allow an unchecked subexpression to acquire an arbitrary type.

Example 11.3 — Cheap contradictions and unresolved arithmetic. *Base fragment+$\mathcal{K}$*

```
eager: {x: <0, y: x & >0}    // refuted by constant-bound intersection

hypothesis: {
    a: int & b * 5
    b: a - 1
}                            // retained without symbolic elimination
concrete: hypothesis & {a: 0} // refuted: b = -1 and a = -5
```

The arithmetic relation has no integer solution: $a = 5(a - 1)$ implies $a = 5/4$. Its general emptiness need not be decided. Type checking establishes the kinds of multiplication and subtraction, not the satisfiability of the equations. Grounding a particular instance enables exact refutation. This matches the distinction between incomplete cyclic constraints and concrete evaluation in CUE [24].

The corpus requirement concerns the eager service and inputs available at the same checking stage. An observation that needs future runtime data is checked when those data arrive. An implementation cannot claim compliance by running no transfer rules or by exhausting its budget before every trivial judgment. Resource exhaustion blocks acceptance and is reported separately.

11.5 Relevant consistency of explicit descriptions

An explicit interface describes successful observations on the input regions it mentions. Arrow membership constrains partial computations; relevant consistency checks whether that explicit description has already refuted a stated successful observation. These are separate judgments. A relevance diagnostic blocks source admission without equating the denotation of a function type to bottom.

11.5.1 Relevant domains and pointwise result specifications

For a finite conjunction of explicit arrow clauses

$$C = \bigwedge_{i \in I} (A_i \Rightarrow B_i),$$

let p range over complete call packets. Its *relevant domain* is $D_C = \bigcup_{i \in I} A_i$. Its partial result specification is

$$\mathcal{B}_C(p) = \bigcap_{\{i \mid p \in A_i\}} B_i \qquad (p \in D_C).$$

Outside D_C this specification is undefined: there is no relevant observation there. Undefinedness is neither a top result nor a failed computation.

For each nonempty set $J \subseteq I$, define the exact input region and its combined result predicate by

$$R_J = \left(\bigcap_{j \in J} A_j \right) \setminus \left(\bigcup_{i \in I \setminus J} A_i \right), \qquad B_J = \bigcap_{j \in J} B_j. \tag{11.2}$$

The R_J partition D_C; $\mathcal{B}_C$ is B_J on R_J. With dependent results, retain $B_j(p)$ in the same packet context. Calling conventions are part of A_i, so arguments with different arities or incompatible required labels need not inhabit overlapping regions.

Let $\Gamma \vdash_{\mathcal{K}} P \equiv \bot$ denote a verified refutation by the finite eager service and the selected strict proof rules. Failure to establish this judgment is not a satisfiability proof. Complete the mandatory eager pass before classifying a region; resource exhaustion blocks the check.

Definition 11.1 (Relevant consistency)**.** *An explicit arrow conjunction C fails relevant consistency in declaration context Γ when a generated nonempty index set J satisfies*

$$\Gamma \nvdash_{\mathcal{K}} R_J \equiv \bot, \qquad \Gamma, p : R_J \vdash_{\mathcal{K}} B_J(p) \equiv \bot. \tag{11.3}$$

A relevance certificate records the prescribed region checks and the absence of such a refuted relevant observation. The predicate $\mathrm{RC}_{\mathcal{K}}(\Gamma, C)$ denotes passage of this finite check.

For nondependent results the second judgment can be checked in Γ alone. A refuted input region is discarded before checking its result; inconsistency of the region must not be used as an ex-falso proof of a live result conflict. A domain that is merely unresolved is subject to the check.

This is a source well-formedness condition relative to a published refutation kernel. Passage asserts neither totality nor satisfiability of every result predicate. A body may still fail on particular inputs, or on all inputs. A failure-only predicate synthesized for that body remains a valid internal type.

Example 11.4 — Refuted explicit observations. $\mathcal{U}_1 + \mathcal{K}$

```
f: (func(int) -> int) & (func(int) -> bool)
// blocked: on int, the explicit result is int & bool

g: func(int) -> _|_
// blocked: an explicit empty result on a non-refuted domain

h: (func(number) -> int) & (func(int) -> bool)
// blocked on int, even though the other numeric region is consistent
```

The first two declarations have the same relevant result specification. In the third, every overlapping integer input requires an impossible successful result. Successes elsewhere in the domain, or outside all declared domains, do not satisfy that observation.

An explicit empty result is rejected by equation (11.3); the internal empty type, executable failure terms, and empty inferred results remain part of the calculus. CUE's bottom expression retains its ordinary evaluation role [1].

Example 11.5 — Nonconflicting overlap and disjoint cases. $\mathcal{U}_1+\mathcal{K}$

```
a: (func(number) -> number) & (func(int) -> int)
// accepted interface: integers on int, numbers on other numeric inputs

b: (func(int) -> string) & (func(string) -> int)
// accepted interface: the two domains are disjoint

c: (func(int) -> (0 | 1)) &
   (func(int) -> (1 | 2)) &
   (func(int) -> (0 | 2))
// blocked: all three results have empty intersection on int
```

The third example requires joint conjunction, not merely pairwise consistency: each pair of its result predicates intersects. Equal-domain grouping exposes the contradiction without enumerating input values.

11.5.2 Source boundaries, rigid binders, and instantiation

A *relevance root* is the accumulated explicit type description of one subject at one declaration boundary. Inline and independent annotations share that root. It carries source origins, quantifier dependencies, and guards. Parametric aliases expand before checking and retain the origin of their use. An inferred type, a proof intermediate, or a selected generic instance creates no additional root.

At a universal declaration, check its body under fresh rigid type variables and the declared bounds. Refutation of a result must follow under those variables; the checker does not choose a type assignment solely to make the result empty. This convention permits declarations whose individual instances may fail.

Example 11.6 — A generic computation and an explicit specialization. $\mathcal{U}_1+\mathcal{K}$

```
meet(A, B): func(x: A, y: B) -> (A & B): x & y
// accepted: A & B is not refuted at the declaration

good: meet({a: 1}, {b: true}) // {a: 1, b: true}
bad:  meet[int, bool](1, true) // _|_, a failing computation

specialized: func(int, bool) -> (int & bool)
// blocked: a newly declared interface with refuted results
```

The instance $[\mathbb{Z}, \mathsf{Bool}] \Rightarrow \bot$ in the call is derived from an accepted generic declaration. It is an internal description of the computation, not a new assertion that this particular interface describes successful behavior. Instantiation and inference must not convert it into an explicit annotation and rerun relevance. An annotation written by the programmer does create a root, including one written through a parametric alias.

Conjoining a universal scheme with another explicit arrow also requires checking their stated overlapping obligations. The checker uses the non-generic domains of sibling clauses as finite *anchors*. Match a universal clause's domain pattern against an anchor, verify the resulting substitution and bounds, and add that instance to the root's observation checks. Only variables bound by that universal may be substituted; outer existential dependencies remain correlated variables. The minimal rule uses finite structural matching, with repeated variables required to match the same type. Additional certified matches may be supplied within the same budget.

Example 11.7 — A universal obligation at an explicit sibling domain. $\mathcal{U}_1+\mathcal{K}$

```
quiet(A): func(A) -> A
quiet:    func(int) -> bool
// blocked before any body is supplied
```

The integer domain anchors the instance $A = \mathbb{Z}$, giving the conflicting results $\mathbb{Z}$ and Bool. Matching is

triggered by the independently stated integer interface, not by a call. The generic `meet` declaration has no such independently stated ground-domain anchor. Its later applications do not modify its source root.

Alpha-renaming, quantifier floating, and copied declarations retain which variables a source clause depends on. An integer clause floated under `forall A` remains independent of A and remains an anchor. Equivalent same-domain result splitting and conjunction regrouping generate identical relevance checks. Arbitrary substitution of an accepted generic declaration by a newly written specialized interface need not preserve source admissibility; its denotation still obeys the substitution lemma.

11.5.3 Nested observations, alternatives, and guards

Traverse explicit parameter, result, record, and list interfaces recursively. At each function-valued subject, collect conjunctive arrow requirements before forming regions. Required-field and element-presence guards are inherited from the enclosing description. An impossible optional field constrains absence; an empty-list alternative makes no element observation.

Union alternatives describe different possible interfaces. Check each surviving explicit alternative separately; never intersect arrow results from opposite sides of a union. A guard proved impossible by ordinary eager normalization removes its branch before relevance traversal. A semantically inhabited but relevance-invalid arrow alternative cannot be silently discarded as though it were an empty data alternative.

Example 11.8 — Separate alternatives and nested callback contracts. $\mathcal{U}_1+\mathcal{K}$

```
choice: (func(int) -> int) | (func(int) -> bool)
// accepted interface: alternatives, not simultaneous obligations

callback: func(
    (func(int) -> int) & (func(int) -> bool),
) -> string
// blocked: the explicit callback interface is relevance-invalid
```

The source checker validates stated observations even in negative type positions. This is a formation condition, independent of the semantic variance of that position.

For dependent results use the packet context in equation (11.3). For example, an integer parameter x with result constraints $y > x$ and $y < x$ has refuted successful results whenever the selected arithmetic rules establish the contradiction. Unresolved arithmetic remains unresolved; relevance does not supply a general satisfiability oracle.

11.5.4 Finite checking and preservation properties

A straightforward algorithm enumerates the regions of equation (11.2), discards refuted guards, and intersects the remaining result predicates. A shared decision DAG avoids duplicating equivalent regions. Equal-domain aggregation, disjoint-kind tests, finite literal sets, and constant interval intersections are mandatory fast cases. Nonempty subsets of three or more clauses must be considered when their cumulative results can conflict.

A conflicting subset can guide the search. For $J \subseteq I$, put $Q_J = \bigcap_{j \in J} A_j$. If its result conjunction is refuted, every surviving refinement of that overlap has the same conflicting requirements. Split Q_J by the remaining domain guards and stop at a non-refuted exact region. A certificate retains that region, its contributing source origins, and the result refutation. A known inhabitation witness for Q_J can identify a live region directly. This search need not enumerate unrelated regions; unknown overlap alone is not a proof that any particular exact region survives.

All expansion, matching, guard checks, and region splitting consume the static budget. Exhaustion blocks admission; it never omits a required relevance check or turns it into a runtime assertion. On a finite closed kind/literal partition, the reference procedure decides relevant consistency exactly. Richer predicates use the published finite refutation relation. The existing rank profiles and $\mathcal{K}$ remain

orthogonal: every profile applies relevance at explicit interface boundaries, while their available match and proof forms differ.

Proposition 11.2 (Relevance coherence)**.** *For fixed source roots, scopes, and canonical checking rules, permuting or duplicating interface conjuncts preserves relevant consistency. Replacing a same-domain arrow with the conjunction of its result-split arrows preserves the relevant regions and their result predicates. These transformations do not change erased code.*

Proof. The relevant domain is a union of declared domains; each relevant result is an intersection of all applicable requirements. Both operations are associative, commutative, and idempotent. Result splitting repeats the same guard, and alpha-renaming preserves its dependencies. Canonical sharing makes the prescribed comparisons and their budget charges independent of grouping. The relevance pass constructs evidence or diagnostics, never executable nodes. □

Relevant consistency is a source admission condition, not a predicate equality in $\mathcal{P}(V)$ and not an additional lattice operation. Adding a declaration can expose an invalid relevant observation while leaving the denotation inhabited. If a conflicting region has an inhabitation witness, further arrow conjuncts cannot remove the conflict: they only refine results there. When the region is merely non-refuted, the blocking judgment is relative to its retained guard and checking context; it is not a proof of semantic inconsistency. The normalizer therefore never turns a relevance diagnostic alone into bottom. Denotational refinement and annotation erasure retain their existing theorems.

11.6 The static–dynamic boundary

Annotations and executable assertions have different judgments. An annotation T requires a derivation of the body's successful-result inclusion in T. The expression `e & P` executes a constraint operation written by the programmer. When P has a checked first-order interpretation, use

$$\frac{\Gamma \vdash e \Rightarrow S \qquad \Gamma \vdash P\ \mathsf{predicate}(K) \qquad \Gamma \vdash S \leq K}{\Gamma \vdash e\ \&\ P \Rightarrow \{v \in S \mid P(v)\}}. \tag{11.4}$$

Here K is the predicate operation's admissible data interface. A predicate with ordinary CUE meet semantics on all kinds, such as a kind constraint, can use $K = \top$ and fail on incompatible kinds. Operation checking is static; success or failure of the explicit assertion may depend on runtime data.

The predicate rule and the ordinary meet rule are source primitives, not casts introduced by a subsumption rule. For `x: int`, the body `x & >0` synthesizes $\mathbb{Z} \cap [\![> 0]\!]$. Checking that result against the same bound is reflexivity. Checking the body `x` against that bound instead requires $\mathbb{Z} \subseteq [\![> 0]\!]$ and is blocked.

The minimum entailment kernel includes conjunction elimination/introduction, reflexivity of alpha-equivalent dependent predicates with the same captured dependencies, constant-bound inclusion, and the strict rules above. These rules accept explicitly asserted refinements without a general arithmetic theorem prover. More involved implication proofs may be supplied or searched within a budget. An unproved implication always blocks; a runtime assertion must already occur in the source term.

A source predicate can retain a pending execution exactly when:

1. its operands and data interface have been checked;
2. its successful-result rule is certified independently of the requested annotation;
3. its executable predicate and all shared dependencies remain in the source evaluation graph;
4. the eager service has not already proved its completion predicate empty in the applicable scope.

A predicate returns pass, fail, or pending. Only pass releases a completed value with its refinement. A refuted predicate is normalized to failure instead of remaining pending. A predicate not refuted by the finite abstraction is a possible residual, not a proved satisfiable relation.

Bounds, regular expressions, finite data traversals, and comparisons on actual arguments admit such rules. Quantified function behavior, impredicative membership, and opaque representation membership require static evidence; their mere occurrence in an annotation supplies no executable test.

Example 11.9 — An annotation and an explicit assertion. $\mathcal{U}_1+\mathcal{D}+\mathcal{K}$

```
positive: func(x: int) -> (int & >0): x
// blocked: int does not imply >0

positiveChecked: func(x: int) -> (int & >0): x & >0
// accepted: the body explicitly establishes the refinement on success
ok:  positiveChecked(3)   // 3
bad: positiveChecked(0)   // _|_
```

The accepted body has the same assertion before and after certification. Its annotation erases; its `& >0` operation remains.

Example 11.10 — Explicit kind refinement. $\mathcal{U}_1+\mathcal{K}$

```
pick: func(x: int | string) -> int: x
// blocked: the result inclusion is false

pickChecked: func(x: int | string) -> int: x & int
// accepted: successful unification produces an integer

same(A: number): func(x: A) -> A: x + 1
// blocked: a numeric upper bound does not establish closure of A
```

The source assertion in `pickChecked` can reject a string. A type annotation on `pick` cannot introduce that rejection.

11.7 Conjunctive declarations and body evidence

For each subject, collect a scoped conjunction of its interface predicates and its implementation clauses. Inline and separate function-type annotations enter the same conjunction. Their placement, order, and parenthesization have no operational significance. A concrete implementation contributes its original body, captured environment, and call protocol.

Let C_f be the finite set of interface conjuncts for f, and let e_f be its body. First check relevant consistency of the accumulated explicit root, irrespective of whether e_f is available. The checker then verifies e_f against $\bigwedge C_f$, checking each universal clause at its rigid instance. It retains the body's inferred refinement or replayable derivation, rather than using a weaker declared result type as its only evidence. A separately compiled opaque import instead exposes precisely its certified interface. A stronger client requirement needs stronger evidence from that module, not a client-side wrapper.

Example 11.11 — Five presentations of one implementation constraint. $\mathcal{U}_1+\mathcal{D}+\mathcal{K}$

Each listing is an independent declaration set for the same field.

```
positive: func(x: int) -> (int & >0): x & >0
```

```
positive: func(int) -> (int & >0)
positive: func(x: int) -> _: x & >0
```

```
positive: func(x: int) -> int: x & >0
positive: func(int) -> >0
```

```
positive: func(int) -> int
positive: func(x: int) -> >0: x & >0
```

```
positive: func(int) -> int
positive: func(int) -> >0
positive: func(x: int) -> _: x & >0
```

All five have the erased body $\lambda x.\,(x \,\&\, [\![> 0]\!])$ and the combined obligation

$$(\mathbb{Z} \Rightarrow \mathbb{Z}) \cap (\mathbb{Z} \Rightarrow [\![> 0]\!]) = \mathbb{Z} \Rightarrow (\mathbb{Z} \cap [\![> 0]\!]).$$

The body synthesis and result inclusion are identical, modulo redundant conjuncts. The relevant integer region has result $\mathbb{Z} \cap [\![> 0]\!]$, which is not refuted. All five roots pass relevant consistency. Each declaration set is accepted; calls at 3 return 3 and calls at 0 fail.

Proposition 11.3 (Declaration coherence)**.** *For a fixed implementation origin and scope graph, regrouping, permuting, or duplicating interface conjuncts preserves the accumulated predicate and erased body. Splitting a result intersection into separate same-domain arrow constraints preserves both and preserves relevant consistency by proposition 11.2. The required kernel rules certify the same result for the five presentations above. Separately written examples are compared up to a consistent renaming of their implementation origins.*

Proof. Predicate conjunction is associative, commutative, and idempotent. The arrow intersection law in proposition 4.1 distributes a fixed domain over result intersection. Body extraction ignores result annotations, and checks all conjuncts against the same retained body derivation. No checking step contributes an executable operation. □

For arbitrary equivalent formulas, bounded proof search need not discover the same derivation. The mandatory syntactic regroupings use canonical conjunction DAGs and shared body evidence, so they neither duplicate the budget charge nor depend on file traversal order.

11.8 Failure-only predicates and their scope

Write $\mathrm{Succ}_\Gamma(e)$ for the successful-completion predicate of e in its retained environment. The finite kernel recognizes empty kind intersections, incompatible constants and bounds, and the empty result of an applicable arrow-clause intersection. A proved empty predicate triggers

$$\Gamma \vdash \mathrm{Succ}(e) \subseteq \varnothing \quad \Longrightarrow \quad e \longrightarrow \_\,|\,\_.$$

This is mandatory whenever the implemented eager rules establish the premise. No supplied body or concrete value is needed to refute an already empty data hypothesis. Required field propagation turns a bottom field into an inconsistent configuration. A lambda body is a different scope: its failure describes a call, not a failure to construct the closure.

For an internal result specification $(A \Rightarrow B) \cap (A \Rightarrow C)$, any correctly admitted call at an argument in A has successful results in $B \cap C$. When that intersection is empty, the call reduces to bottom. This rule applies to derived generic instances and inferred body paths. A newly written interface asserting the same empty relevant result is instead blocked by section 11.5, before a call or implementation is required.

Example 11.12 — Invalid interfaces and scoped computation failure. $\mathcal{U}_1{+}\mathcal{K}$

```
q: func(int) -> int
q: func(int) -> bool        // blocked: explicit relevant conflict

meet(A, B): func(x: A, y: B) -> (A & B): x & y
observed: meet(1, true)     // _|_: computation, not interface rejection

stop(A): func(x: A) -> A: _|_
// a closure with a failing body and a non-refuted generic interface
```

Type checking still verifies subexpressions and calling protocols before using an empty-result rule. An ill-typed operation is not repaired by assuming that some unrelated computation fails.

11.9 Stages and mandatory certificates

Stage	Obligations completed before advancing
Elaboration	Scope and sort correctness; profile restrictions; alias expansion; fixed calling protocols; accumulation of interface conjuncts and identification of explicit source assertions.
Interface admission	Relevant consistency of every explicit root, including unimplemented imports and nested callback types; guarded region checks and source-anchored universal matching.
Definition checking	Every function body, including uncalled bodies, has a strict derivation. Prove every result implication using the unchanged body, including the successful-result rules of its explicit assertions.
Linking	Every supplied user function, builtin, foreign adapter, and captured callable satisfies the required strict interface. Unspecified imports remain link obligations and prevent a closed executable artifact.
Call checking	Check every statically present application and type application. Prove packet compatibility and every argument/result implication. Source assertions in arguments may supply the required refinements. Generic instances are derived types, not new relevance roots. Newly visible literal and shape contradictions are propagated at their scopes.
Normalization	Run eager transfers and exact enabled primitives; maintain joint constraints, alternatives, guards, and source-assertion dependencies. Propagate established empty predicates at their scopes.
Concrete observation	Demand and discharge every source assertion on the observed path and every required data-validation dependency. Failure produces bottom; pending conditions prevent unconditional completed export.

Modules may store relevance-checked unlinked declarations as hypotheses. Their certificates are conditional on named imports, as a typing derivation is conditional on its context. A body or import does not become certified merely because no call currently observes it. Calling an unknown function does not synthesize a suitable implementation from the desired codomain.

Example 11.13 — Higher-order obligations are never runtime residuals. $\mathcal{U}_1+\mathcal{K}$

```
apply: func(g: func(number) -> string) -> string: g(1.5)
ints: func(x: int) -> string: "ok"
blocked: apply(ints)         // blocked before execution

source: func(int) -> string // an unresolved external implementation
client: func() -> string: source(3)
```

The strict contravariant rule rejects `ints` as the required callback. The semantic possibility that its out-of-domain calls fail does not discharge that rule. This is the same intentional separation between denotational safety and a useful syntactic type discipline that prevents arbitrary dead code from being accepted solely on a conjecture that it never returns.

11.10 A bounded implementation

Store types, interface conjunctions, and body summaries as hash-consed DAGs; identify binders and implementations by stable origins. Store proof DAGs separately from source evaluation nodes. The checker for one accumulated subject has the following form:

```
check_subject(declarations, budget):
    scoped = canonicalize_scopes_and_conjuncts(declarations)
    body, interface, hypotheses = separate_code_and_predicates(scoped)
    state = eager_normalize_with_scoped_failure(
        conjunction(hypotheses, interface), budget)
    if state.required_subject_is_empty:
        return Refuted(state.proof)
    relevance = check_explicit_relevance(scoped.roots, state, budget)
    if relevance is invalid or relevance is exhausted:
        return Blocked(relevance.diagnostic)
    if body is absent:
        return Hypothesis(state, interface, relevance)
    source = elaborate_source_operations(body)
    evidence = check_body_and_explicit_assertions(source, budget)
    for clause in canonical_clauses(interface):
        proof = check_implication(evidence, clause, budget)
        if proof is absent:
            return Blocked(clause, reason="annotation not established")
        verify_proof(proof, budget)
    return Accepted(erase_annotations(source), evidence, relevance, state)
```

Elaborating a source assertion lowers an operation already present in the term; it never consults the requested output predicate to invent a check. A missing body remains an ordinary hypothesis, not a certified implementation. Linking a body reruns the same interface judgment. A retained source operation may await its operands only after its operation rule and annotation implications have been checked.

All work is charged, including alias expansion, substitution, arithmetic, and Boolean splitting. Canonicalized duplicates are shared. Published budgets and deterministic traversal make required regroupings reproducible. Budget exhaustion blocks certification; it does not alter code or create a runtime obligation. Source evaluation has its own resource policy and cannot report a false validation pass.

Let L be the size of a supplied proof DAG after sharing and let C bound its charged local comparisons. Verification costs $O(L\,C)$ time and $O(L)$ graph space. Search uses a declared fuel B, costing at most $O(B\,C)$ work. Unions and intersections can require exponentially many alternatives. The kernel promises its finite rules and minimum fixtures, not complete semantic inclusion at a fixed budget.

The minimum direct rules cover kinds, constants, finite record lookup, tuple projection, explicit type substitution, meet, predicate reflexivity, and the five declaration presentations. In particular, explicit `x & >0` needs no proof that all inputs satisfy `>0`. Its primitive successful-result rule supplies the fact needed by annotation checking. Explicit integer-to-integer and integer-to-boolean conjuncts are rejected by relevance before any body check. A pass-through implementation that stores such an interface as a hypothesis is not conforming.

11.11 Erasure and refinement preservation

Let $|e|$ erase function-type annotations, type abstractions, type-instantiation selections, and proof arguments. It preserves the source's data constraints, term-level meets, predicate operations, captures, labels, omission behavior, and computational indices. Representative clauses are

$$
\begin{aligned}
|\,\mathrm{func}(x : S) \to T : e| &= \lambda_{protocol} x.\,|e|, \\
|\forall A\; e| &= |e|, \qquad & |e[T]| &= |e|, \\
|e \;\&\; P| &= |e| \;\&\; |P| \qquad & &\text{for an executable assertion.}
\end{aligned}
$$

The same predicate spelling can occur in an annotation or as an operand of a source operation. Its elaborated role determines erasure: `-> >0` is a proof requirement; `x & >0` computes a constrained result. Ordinary CUE field constraints remain part of the data language, including their validation dependencies. They are not discarded as function-type annotations.

Theorem 11.4 (Annotation erasure)**.** *Assume the source-operation rules preserve their specified observations and the static kernel is sound. For an Accepted implementation in an environment satisfying its certified interface, erasing its annotations preserves successful returns, explicit assertion failures, and the retained demand dependencies. The erasure is independent of the placement and grouping of its annotations.*

Proof. Each typing rule contributes evidence and no executable filter. Lambda erasure retains the binder's operational protocol and the erased body. Instantiation changes only the proof environment. Subsumption and intersection introduction retain the same term. Relevant-consistency checking contributes only static evidence or a blocking diagnostic. Executable meets and assertions have the same evaluation rule on each side of erasure. Induction on evaluation, together with proposition 11.3, gives observation preservation and declaration independence. Computational witnesses, sealing operations, and omission defaults remain terms and follow their specified source rules. □

For scoped declarations D and E sharing an interface, the elaborated predicate is refinement-homomorphic:

$$[\![D \,\&\, E]\!] = [\![D]\!] \cap [\![E]\!]. \tag{11.5}$$

For a common body origin, the executable contribution on both sides is the same erased body; interface conjunction adds predicates and proof obligations. An incompatible additional annotation refutes a predicate, fails relevant consistency, or blocks its conformance certificate. It never edits execution to enforce the new annotation. Distinct body origins retain the closure-identity semantics in chapter 7. Equation (11.5) concerns declaration predicates; application still satisfies the inclusion (4.2), not a general meet-homomorphism equation.

Keep the exact residual predicate and its scope graph during normalization:

$$P \longrightarrow Q \quad \Longrightarrow \quad P \simeq Q, \qquad P \wedge R \simeq Q \wedge R. \tag{11.6}$$

A source assertion passing on a value establishes its refinement. Unresolved captured dependencies remain attached to that success; for example, `3 & (x + 1)` cannot become unconditional `3` while `x` is unresolved. A proved empty required predicate propagates bottom. Quantifier and branch scopes prevent a local body failure from becoming a false global refutation.

Theorem 11.5 (Static gate and explicit-assertion partial correctness)**.** *Assume sound strict rules, sound scoped eager refutation, dependency-preserving normalization, and the source assertion rules. Every Accepted artifact has a certificate for every annotation and has passed the required relevance checks at each explicit interface root. In an environment satisfying its imports, every successful completed observation has its stated type and demanded refinements. Further conjunction cannot repair a proved data contradiction or replace an atomic completed observation by a different atom. Erasure preserves the implementation's observations.*

Proof. The admission gate establishes the relevance-check clause; it contributes no additional execution rule. Induction on body and implication certificates gives theorem 10.1. The source assertion rule constrains successful results without a proof of success. Equation (11.6) preserves exact constraints, and theorem 3.1 lifts this preservation through quantification. A subset of an empty predicate is empty; a subset of a singleton either retains that atom or is empty. Demand dependencies prohibit unconditional release of a conditional value. Theorem 11.4 supplies the final assertion. □

11.12 Profile combinations and conformance examples

Combination	Mandatory checking organization
$\mathcal{U}_1$+$\mathcal{K}$	Prenex schemes, monomorphic argument interfaces, finite instance inference; explicit instances where local search is insufficient.
$\mathcal{S}_1$+$\mathcal{K}$	The preceding kernel plus module-boundary existential witnesses, sealing, and shared rigid openings.
$\mathcal{S}_H$+$\mathcal{K}$	Annotated arbitrary-rank types and explicit impredicative instantiation. Quantified membership never becomes a runtime predicate.
$\mathcal{D}$+$\mathcal{K}$	Mandatory strict skeleton; eager known dependent contradictions; source assertions supply certified refinements, and every annotation implication is proved.
$\mathcal{A}$+$\mathcal{K}$	Strict abstract operations and scope checks. Representation independence additionally uses an outcome-preserving simulation.

The following are minimum conformance obligations, not optional examples:

Judgment or program	Required outcome
$(1 \vee 2) \leq \mathbb{Z}, \mathbb{Z} \leq \mathsf{Number}$	Proved statically.
$(1 \vee \mathsf{true}) \leq \mathbb{Z}$	Blocked; no inserted kind check.
$\{a : 1, b : \mathsf{true}\} \leq \{a : 1\}$	Proved statically.
$\{a : 1\} \leq \{a : 1, b : \mathsf{Bool}\}$	Blocked required-field judgment.
Identity at $\forall A.\, A \Rightarrow A$	Accepted with a rigid-variable proof.
Identity instantiated at its own quantified type	Accepted in $\mathcal{S}_H$ with explicit instantiation.
`func(xs: [A,A]) -> A: xs[0] & xs[1]`	Accepted by tuple projection and meet.
An integer result annotated as string	Blocked at definition.
An integer callback used where a number callback is required	Blocked at the static call or link boundary.
An explicit ground interval contradiction	Refuted before execution.
An unresolved arithmetic cycle with compatible operand kinds	Retained as data constraints; ground instances checked.
A dependent result bound absent from the body evidence	Blocked unless its implication is proved.
`x & >0` at `int & >0`, under `x: int`	Accepted by the explicit assertion rule.
The five presentations in section 11.7	Accepted with one erased body and equivalent contracts.
Explicit $\mathbb{Z} \Rightarrow \mathbb{Z}$ and $\mathbb{Z} \Rightarrow \mathsf{Bool}$ on one subject	Blocked by relevance without a body or a call.
An explicit $\mathbb{Z} \Rightarrow \bot$	Blocked by relevance.
A conflicting overlap with a consistent region elsewhere	Blocked; other regions do not repair the overlap.
The generic meet declaration with rigid A, B	Accepted; $A \cap B$ is not refuted at the declaration.
A failing derived instance of generic meet	Reduced to bottom at its call scope; no new explicit-interface relevance check.
An admitted derived call with a proved empty result intersection	Reduced to bottom at the call scope, before implementation linkage if possible.
An unresolved polymorphic conformance judgment	Blocked, never a dynamic residual.
Static resource limit exceeded	Blocked with a resource diagnostic.

Chapter 12

Feature correspondence and related designs

12.1 The comparison baseline

The July 2026 function proposal describes opaque function values, packet binding, defaults, open signatures, and partial applications. The September theory addendum analyzes a covariantly tightened call-constraint interpretation and pointwise conjunction of bodies. It separately proposes coverage obligations and compatibility checking [5, 6].

The present calculus expresses call restrictions existentially and reusable capabilities universally. Its set lattice therefore includes both relations on actual call cells and predicates on implementations. Their different variance follows from their quantifiers.

12.2 Feature coverage

Feature	Interpretation or construction in quantified CUE
Legacy records, lists, unions, bounds	Unchanged data predicates and legacy elaboration; theorem 5.2.
Definitions, closedness, optional and required fields	Retained field-presence and label constraints; quantifiers do not add fields.
References, embedding, comprehensions	Existing copying plus capture-avoiding binder and witness rebinding.
Literal bodies and inferred result constraints	Operational closures with a checked declared or inferred result predicate.
Positional and labeled arguments	Complete call packets and a fixed binder.
Required, optional, hidden, and anonymous parameters	Presence alternatives, package-qualified labels, and positional slots.
Body-only aliases	Lexical aliases with no public packet label.
Parameter order and uniqueness	Protocol well-formedness before value unification.
Parameter attributes	Retained metadata, interpreted only by the relevant declared extension.
Omission defaults	Implementation-owned packet completion; supplied arguments bypass them.
Caller disjunction defaults	Existing preference rules on the supplied value.
Default contributed by another declaration	Explicit defaulted adapter or a refinable description of a yet-unselected implementation protocol.
Label contributed to an existing closure	Explicit label adapter; fresh builtins may receive their generated binder at construction.
Function types and intersections	Sets of operational inhabitants satisfying all universal arrow clauses.
Open signatures	Existential protocol rows retained until the remaining required packet shape is known.
Higher-order parameters and results	Ordinary arrows; universals specify reuse across types.
Partial application	A new closure with normalized original-slot bindings and a residual contract.
Constraint on a partial value	An ordinary constraint on that residual closure.
Distinct bodies combined pointwise	An explicit body that calls both and unifies their result descriptions.
Self-unification and captures	Stable origins and structurally constrained captured environments.
Validators	Predicates on an existential subject, definable with existing structs.
List-based variadic use	One list-valued packet component, with a generic element type.

Feature	Interpretation or construction in quantified CUE
Nonrecursive execution and cycles	The legacy execution profile; structural recursion is a separate opt-in policy.
Builtins and foreign calls	Typed descriptors with an explicit admitted effect set and the existing bridge.
Compatibility checking	Packet-domain inclusion, output/effect refinement, and module simulations where abstraction applies.
Export, formatting, and round trips	Source export retains binder scopes, closure environments, and bound slots; data export retains its existing supported values.
Memoization and dependency tooling	Scope-preserving graph traversal and dependency-aware caching.

Experimental operations that transform a closure elaborate to explicit transformations. Contract intersection asserts properties of the original closure.

12.3 Three explicit translations

Example 12.1 — Supplying a label. $\mathcal{U}_1$

```
raw: func(_~n: int) -> int: n + 1
named: func(x: int) -> int: raw(x)
a: named(x: 2)              // 3
```

The adapter's packet language admits `x`. Raw positional behavior remains unchanged.

Example 12.2 — Restricting public calls across files. $\mathcal{U}_1$

```
raw: func(x: number) -> string: "\(x)"
call: {arg: number, result: raw(arg)}
call: {arg: int, result: string}

ok:  (call & {arg: 7}).result   // "7"
bad: (call & {arg: 1.5}).result // _|_
```

This is ordinary CUE instantiation and projection. The argument restriction is an existential predicate, while `raw`'s capability remains universal.

Example 12.3 — A first-order dependent result check. $\mathcal{U}_1+\mathcal{D}+\mathcal{K}$

```
legacyBody: func(int) -> int
checked: func(x: int) -> (int & >0): legacyBody(x) & >0
```

The body explicitly constrains the integer returned by `legacyBody`. Its meet rule proves the refined codomain without assuming success of the comparison. The type annotation erases and the source assertion remains. Linking separately verifies `legacyBody`'s integer interface. Omitting the source assertion requires an independent proof of its positive-result property.

Proposition 12.1 (Contract intersection)**.** *Under the universal arrow interpretation, conjoining a contract with a concrete closure cannot change its call protocol or result. It preserves the singleton closure or makes its intersection empty.*

Proof. A subset of the singleton $\{f\}$ is either $\{f\}$ or empty. □

Consequently, constructing a label/default adapter and asserting a contract are different operations. Keeping that distinction also preserves associativity. A full experimental compatibility layer can implement its original invocation relation as a checked adapter, while translating its public guarantees into the appropriate quantified predicates. Strict kind and callable checks are verified at that boundary rather than silently deferred by the translation.

12.4 Modular guarantees

Universal constraints express a statement the call-restriction interpretation does not express by itself: one selected implementation preserves its result type at every admitted instance whenever that instance returns successfully. This supports checking `map`, polymorphic callbacks, and bounded refinement-preserving utilities without seeing their eventual calls. The intersection algebra preserves input/output correlations. Dependent universals state relationships across argument and result values. Existential seals support representation-independent replacement under an explicit theorem.

Untyped callbacks and struct encodings can express many of the same computations. Quantified descriptions additionally preserve their relationships as constraints.

12.5 CDuce and semantic subtyping

Frisch, Castagna, and Benzaken develop semantic subtyping with function and Boolean types; Castagna and Xu extend its foundation to polymorphic types and convex models [11, 12]. CDuce's polymorphic calculi combine instantiation with intersections and preserve input/output correlations [13, 14]. These supply close precedents for the arrow and instantiation layers here.

The current proposal adds explicit quantifiers to the general CUE predicate language. Its partial-correctness arrows refer to a fixed operational inhabitant model, with impredicative quantification over sets of inhabitants. A subtyping algorithm proved for a different arrow model must be revalidated for the selected reference fragment. Convexity of a semantic subtyping model and relational parametricity are separate properties.

12.6 Elixir and checked dynamic execution

Elixir's set-theoretic work uses unions, intersections, refinements, and guarded function behavior. The strong-arrow account pays particular attention to behavior outside the input domain and interaction with dynamic values [17]. It is relevant to CUE's bridge and legacy-expression boundaries: failure behavior deserves an explicit interpretation alongside the successful result type.

This proposal's universals are general constraint binders. Its partial-correctness and optional effect arrows are specified by chapter 4, rather than identified with Elixir's strong-arrow construction. Both approaches use set-theoretic descriptions to retain more precise information than an uncorrelated union of inputs and outputs.

12.7 Hyperdoctrines and compact closure

Consider objects that are finite interfaces and morphisms that are definable relations $R \subseteq X \times Y$. Composition is

$$(S \circ R)(x, z) \equiv \exists y.\ R(x, y) \wedge S(y, z).$$

Juxtaposition of interfaces is the tensor. Equality gives the unit/counit relations making every object self-dual; the snake identities follow by existentially eliminating equality witnesses. This is the compact structure of a relational category.

Adding universal formulas enriches the class of definable relations while retaining these operations, provided the fragment is closed under the required composition and equality relations. Conjunction within one predicate fiber is distinct from the tensor of interfaces. Executable functions form a separate operational subcollection. The compact structure belongs to the relations and does not require executable total functions.

Lawvere's adjunctions and the bifibrational account of Melliès and Zeilberger organize these refinement predicates over the relational interface category [7, 8].

Chapter 13

Requirements and implementation outcome

The addendum's section 9 lists six requirements. The last is its restriction on expressive strength, referring to section 7. The first five are evaluated below against the defined reference semantics [6].

13.1 Directionality

A reusable capability is a universal predicate on the implementation. A particular call is a relation on existential argument and result cells. Neither judgment guesses whether a field is provided or consumed. A higher-order parameter introduces its required capability in the caller's type. Module replacement uses its explicitly declared interface and observation relation.

13.2 Monotone refinement

Conjunction refines the joint predicate. Both quantifiers preserve that refinement, by theorem 3.1. Dependency-preserving copying and retention of bound variables prevent accidental weakening of universal obligations. An established contradiction remains empty, and an atomic concrete result can only persist or conflict. Chapter 11 implements this property by retaining mandatory annotation certificates, explicit source assertions, and validation dependencies. Relevant-consistency errors block explicit interfaces; they do not rewrite inhabited function predicates to bottom. Annotation erasure preserves the source body, and its declaration predicates obey equation (11.5). Defaults and comprehension execution retain the separate observation treatment already identified in the reference model.

13.3 Cross-file input and output invariants

Existing argument/result records express existential call restrictions. Universally quantified declarations express reusable behavior and dependent relations. Both can be conjoined across files. chapters 7 and 8 give concrete examples of each form, including length preservation and request-indexed responses.

13.4 Callable self-unification

Function inhabitants have stable semantic origins and captured environments. Their equality is structural on these descriptors, rather than pointer-based. Conjunction is idempotent on their singleton predicates, including closures reached through copying or unrelated record refinement. Partial applications use normalized original-slot bindings. Sealed operation handles preserve the interface's public alias structure.

13.5 Idiomatic foreign interfaces

Portable logical domains remain in public signatures. Representability and conversion failures may become ordinary bottom; an explicit effect records them when their tags are observable. The existing bridge performs its existing checks and reports those outcomes. Logical kind/domain mismatches are distinguished from in-domain foreign failures. Native widths remain private bridge metadata. No additional conversion stage is required by quantification.

13.6 Expressive strength and implementation milestones

The reference semantics admits higher-rank and dependent specifications. An execution profile may retain the current nonrecursive policy or admit recursion with partial-correctness typing. Neither checking nor arrow membership requires a termination proof. $\mathcal{U}_1$ is an implementation entry point with explicit generic capabilities and semantic arrow intersections. $\mathcal{S}_1$ adds abstract type witnesses at module boundaries. $\mathcal{S}_H$ extends the same formation rules to arbitrary well-sorted positions.

A conforming implementation establishes the following contracts:

1. Elaboration preserves legacy data observations, lexical binding, copied witness sharing, and quantifier dependencies.
2. Every explicit interface passes relevant consistency, independently of implementation linkage. The checker verifies every mandatory strict judgment before accepting executable code. Only explicit source operations and data hypotheses may remain pending; unproved annotation implications block certification. Proved empty predicates normalize to bottom at their retained scopes.
3. Symbolic evaluation preserves the successful-completion semantics and the specified observation/-effect behavior.
4. The abstraction boundary preserves seal identity, public aliasing, and the equality-respecting simulation used for module evolution.

13.7 Conclusion

Universal and existential binders are dual projections of the same relation-valued constraint semantics. Unification remains intersection. Function types arise by universally constraining an operational application relation; their variance and polymorphic behavior follow from that definition. Dependent contexts and existential modules extend the same structure. The resulting language can be implemented in rank-restricted stages without changing what its quantified descriptions mean. Relevant consistency checks the successful observations claimed by explicit interfaces, while partial-correctness typing preserves ordinary computation failure.

Appendix A

Core judgments and finite regression model

A.1 Formation and binding summary

A reference type expression has the grammar

$$T, U ::= b \mid a \mid T \wedge U \mid T \vee U \mid \neg T \mid T \times U \mid \mathsf{List}(T) \mid T \Rightarrow U \\ \mid \{\ell_i : T_i\}_{i \in I} \mid \forall a : S\,T \mid \exists a : S\,T.$$

Here b is a base predicate; a is a context variable of the appropriate sort; S is a well-formed value sort or the type sort Type, possibly with a guard. Complement is relative to the subject sort. A surface profile may retain only its supported decidable negative predicates. Dependent descriptions extend T_i with references to earlier argument or record cells, using the shared binding graph. Function effects refine the application predicate of chapter 4.

Contexts are telescopes. The free variables of a binder's sort and guard must already occur in its context. A bound variable is identified by its binder, not by its spelling. A quantified expression keeps its subject outside the new binder; an explicit dependent sum keeps the witness as part of the subject.

A.2 Sound introduction and elimination principles

Write $\Gamma; \Delta \vdash e : T$ for a typing judgment in a term environment Γ and a rigid constraint context Δ. The type variable a is fresh for Γ in universal introduction:

$$\frac{\Gamma; \Delta, a : S \vdash e : T}{\Gamma; \Delta \vdash e : \forall a : S\,T} \qquad \frac{\Gamma; \Delta \vdash e : \forall a : S\,T \qquad \Delta \vdash s : S}{\Gamma; \Delta \vdash e : T[s/a]}.$$

The first premise checks the same executable expression at an arbitrary admissible assignment. Bounds appear as assumptions in $\Delta, a : S$.

For transparent existential descriptions, introduction chooses a witness and elimination checks the continuation at an arbitrary opened witness:

$$\frac{\Gamma; \Delta \vdash e : T[s/a] \qquad \Delta \vdash s : S}{\Gamma; \Delta \vdash e : \exists a : S\,T}.$$

Elimination preserves the subject and shares the witness across its projections. Its result may not expose the local witness without an enclosing existential. Opaque packages additionally use the sealing discipline in chapter 9.

Intersection introduction uses the same term:

$$\frac{\Gamma; \Delta \vdash e : T \qquad \Gamma; \Delta \vdash e : U}{\Gamma; \Delta \vdash e : T \wedge U}.$$

Source-written interfaces additionally pass relevant consistency at their retained roots. Inferred types and instantiated proof conclusions remain in the full semantic type algebra, including empty successful-result predicates. Subsumption uses a checked strict inclusion derivation. Application must pass its static callable and argument judgments. Its conclusion concerns successful results, not termination or absence of bottom. A runtime refinement operation must occur in the source, with its rule checked as in section 11.6. All annotation implications are proved before certification; their proofs erase.

For returned values, the introduction rules give membership in the corresponding intersection or union; computation judgments quantify only over successful returns. Elimination uses the selected instance or a scoped existential witness. These principles specify the reference calculus independently of a particular constraint-solving strategy.

A.3 Regression checks

The source archive includes three executable standard-library Python models.

`checks/check_model.py` enumerates a finite partial-function algebra and checks arrow laws, partial identities, failure-only call regions, quantifier laws, scope conditions, and set-valued meet. A partial identity witnesses the nonempty denotation of the rejected `quiet` interface. The model also checks a finite applicative model whose type domain is the full powerset of its value carrier; quantified types can occur as instance arguments in that model.

`checks/check_kernel.py` implements a small certificate-oriented structural checker and the static/dynamic gate. Its fixtures cover singleton and kind inclusions, record width/depth and closed field lookup, union/intersection rules, explicit impredicative substitution, the generic meet body, blocked annotation obligations, and explicit dependent assertions. Declaration fixtures check regrouping and common erasure for the five presentations. Negative fixtures verify that unproved result bounds, wrong-kind results, and exhausted static budgets produce no executable filters.

`checks/check_relevance.py` checks explicit-domain observation partitions, incompatible overlaps, multiway result conflicts, conjunction coherence, source-anchored universal instances, and the distinction between generic interface admission and derived failure types. Its fixtures also check that relevance rejection is independent of denotational emptiness and produces no executable filter. The exact counts are recorded in `checks/results.json`.

The models test finite instances and selected kernel rules. The implementation contract is the rule set and conformance table in chapter 11; complete integration with CUE's graph evaluator requires the elaboration and demand invariants in chapter 10. In particular, finite tests are not a proof of that integration or of complete semantic-subtyping inference.

Appendix B

Example and feature index

Feature	Location and program forms
Binding syntax	Chapter 5: quantified expressions, declaration parameters, block prefixes, function-local generics, parametric aliases, and instantiation selection.
Pointwise unification	Chapter 3: alpha-renaming, shared subject, constant constraints, guarded bounded clauses, non-distributivity.
Rank-1 programming	Chapter 7: identity, bounded pairing, minimum, map, filter, partition, flat mapping, composition, adapters.
Higher-rank programming	Chapter 7: a polymorphic callback used at two types, returned polymorphic closures, quantified utility records.
Calling conventions	Chapter 7: all parameter modes, defaults, omission, hidden labels, partial application, open protocols, attributes.
Dependent contracts	Chapter 8: exact successor, interval clamp, length-preserving map, safe indexing, append, zip, split, replication.
Dependent structures	Chapter 8: dimension-carrying matrices, transpose, dot product, matrix multiplication, indexed service responses.
Existential reuse	Chapter 9: sealed counters, generic clients, module laws, heterogeneous packages, shared-witness comparison, codecs.
Compatibility	Chapter 12: feature-by-feature correspondence, adapters, old data code, result combination, experimental migration.
Semantic guarantees	Chapters 2 to 4: adjunctions, floating, monotonicity, partial correctness, variance and application.
Mandatory checking	Chapter 11: strict System F/structural certificates, scoped eager refutation, relevant consistency of explicit observations, rigid declaration checks, explicit assertions, declaration coherence, annotation erasure, stage gates, and resource bounds.
Evolution guarantees	Chapter 9: equality-respecting representation simulation, quantified observations, old-compatible views.

References

[1] The CUE Authors. *CUE language specification*. Living specification, accessed 21 September 2026. https://cuelang.org/docs/reference/spec/.

[2] The CUE Authors. *CUE language specification: References*. Specification of copying and rebinding referenced expressions. Accessed 21 September 2026. https://cuelang.org/docs/reference/spec/#references.

[3] The CUE Authors. *Specifying a default value for a field*. Accessed 21 September 2026. https://cuelang.org/docs/howto/specify-a-default-value-for-a-field/.

[4] The CUE Authors. *Package list*. Standard-library interface documentation, accessed 21 September 2026. https://pkg.go.dev/cuelang.org/go/pkg/list.

[5] Marcel van Lohuizen. *Proposal: functions in CUE*. 10 July 2026. CUE proposal 4484, draft at repository revision `a949162aedb86661183c46abef47919d16605014`. https://github.com/cue-lang/proposal/blob/a949162aedb86661183c46abef47919d16605014/designs/language/4484-functions.md.

[6] Marcel van Lohuizen. *Why CUE Functions Tighten Instead of Widen*. Theory addendum to proposal 4484, committed 19 September 2026, repository revision `a949162aedb86661183c46abef47919d16605014`. https://github.com/cue-lang/proposal/blob/a949162aedb86661183c46abef47919d16605014/designs/language/4484-theory-addendum.md.

[7] F. William Lawvere. Adjointness in foundations. *Dialectica*, 23(3–4):281–296, 1969. Reprinted with commentary in *Reprints in Theory and Applications of Categories*, 16:1–16, 2006. https://tac.mta.ca/tac/reprints/articles/16/tr16abs.html.

[8] Paul-André Melliès and Noam Zeilberger. *A bifibrational reconstruction of Lawvere's presheaf hyperdoctrine*. 2016. arXiv:1601.06098. https://arxiv.org/abs/1601.06098.

[9] John C. Reynolds. Polymorphism is not set-theoretic. In *Semantics of Data Types*, LNCS 173, pages 145–156, 1984. https://doi.org/10.1007/3-540-13346-1_7.

[10] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. North-Holland, 1983. https://www.cs.cmu.edu/~crary/819-f09/Reynolds83.pdf.

[11] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4), Article 19, 2008. https://doi.org/10.1145/1391289.1391293.

[12] Giuseppe Castagna and Zhiwu Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *ICFP*, pages 94–106, 2011. https://doi.org/10.1145/2034773.2034788.

[13] Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. Polymorphic functions with set-theoretic types. Part 1: Syntax, semantics, and evaluation. In *POPL*, pages 5–18, 2014. https://doi.org/10.1145/2535838.2535840.

[14] Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. Polymorphic functions with set-theoretic types. Part 2: Local type inference and type reconstruction. In *POPL*, pages 289–302, 2015. https://doi.org/10.1145/2676726.2676991.

[15] The CDuce Project. *Polymorphism tutorial*. Accessed 21 September 2026. https://www.cduce.org/tutorial_polymorphism.html.

[16] Giuseppe Castagna, Mickaël Laurent, and Kim Nguyen. Polymorphic type inference for dynamic languages. *Proceedings of the ACM on Programming Languages*, 8(POPL), 2024. https://doi.org/10.1145/3632882. Preprint: https://arxiv.org/abs/2311.10426.

[17] José Valim. *Strong arrows: a new approach to gradual typing*. 20 September 2023. https://elixir-lang.org/blog/2023/09/20/strong-arrows-gradual-typing/.

[18] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988. https://doi.org/10.1145/44501.45065.

[19] John C. Mitchell. Representation independence and data abstraction. In *POPL*, pages 263–276, 1986. https://doi.org/10.1145/512644.512669.

[20] Jana Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*, pages 429–442, 2013. https://doi.org/10.1145/2500365.2500582. Preprint: https://arxiv.org/abs/1306.6032.

[21] Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. *Scoping constructs in logic programming: Implementation problems and their solution*. 1998. arXiv:cs/9809016. https://arxiv.org/abs/cs/9809016.

[22] Patrick M. Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *PLDI*, 2008. https://goto.ucsd.edu/~rjhala/papers/liquid_types.html.

[23] The GHC Developers. *GHC User's Guide: Arbitrary-rank polymorphism*. Accessed 21 September 2026. https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/rank_polymorphism.html.

[24] The CUE Authors. *Reference Cycles*. CUE language tour, accessed 22 September 2026. https://cuelang.org/docs/tour/references/cycle/.

[25] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977. https://www.di.ens.fr/~cousot/COUSOTpapers/POPL77.shtml.

[26] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. The semantic foundations of concurrent constraint programming. In *POPL*, pages 333–352, 1991. https://doi.org/10.1145/99583.99627.

[27] Paolo Pistone. On completeness and parametricity in the realizability semantics of System F. *Logical Methods in Computer Science*, 15(4), Article 6, 2019. https://doi.org/10.23638/LMCS-15(4:6)2019. Preprint: https://arxiv.org/abs/1802.05143.

[28] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Computing Surveys*, 54(5), 2021. https://doi.org/10.1145/3450952. Preprint: https://arxiv.org/abs/1908.05839.

[29] Shengyi Jiang, Chen Cui, and Bruno C. d. S. Oliveira. Bidirectional higher-rank polymorphism with intersection and union types. *Proceedings of the ACM on Programming Languages*, 9(POPL), Article 71, pages 2118–2148, 2025. https://doi.org/10.1145/3704907.

[30] Jad Elkhaleq Ghalayini and Neel Krishnaswami. *Explicit refinement types*. 2023. arXiv:2311.13995. https://arxiv.org/abs/2311.13995.

[31] Tommaso Petrucciani, Giuseppe Castagna, Davide Ancona, and Elena Zucca. Semantic subtyping for non-strict languages. In *TYPES 2018*, LIPIcs 130, Article 4, 2019. https://doi.org/10.4230/LIPIcs.TYPES.2018.4. Preprint: https://arxiv.org/abs/1810.05555.