TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science / Institute of Computer Engineering, Chair for Compiler Construction

A Processing Pipeline for Querying Bi-temporal Graph Databases

Bilal Mahmoud

bilal.mahmoud@mailbox.tu-dresden.de
Born on: 23.11.1999 in Dresden
Matriculation number: 4843857

Großer Beleg

Supervisors
Felix Suchert

Supervising Professor
Prof. Dr.-Ing. Jeronimo Castrillon

Submitted on: 18.08.2025

Statement of authorship

I hereby certify that I have authored this document entitled *A Processing Pipeline for Querying Bi-temporal Graph Databases* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 18.08.2025

Bilal Mahmoud

Contents

List of Figures

List of Listings

List of Tables

List of Acronyms

AOT. ahead-of-time 18

AST. abstract syntax tree ii, 1, 2, 11, 12, 13, 19, 21, 23, 24, 25, 29, 31, 83, 84, 85, 86, 87, 89, 90, 92, 93

DSL. domain-specific language 11, 12, 14, 15

GLB. greatest lower bound 40, 41, 54, 58

HIR. high-level intermediate representation ii, 11, 21, 23, 24, 29, 31, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93

IH. induction hypothesis 62

IR. intermediate representation 83, 85, 87, 88, 89

IRI. Internationalised Resource Identifier 3

LIR. low-level intermediate representation 91, 92, 93

LPG. labelled property graph 2, 3, 4

LUB. least upper bound 40, 42, 54, 57, 59, 60

MIR. mid-level intermediate representation 91, 92, 93

RDF. resource description framework 3, 4

SCC. strongly-connected components 72

SSA. static single assignment 91, 93

URI. Uniform Resource Identifier 3

VM. virtual machine 23, 88, 91, 92, 93

W3C. World Wide Web Consortium 3

1. Introduction

Graphs have become a mainstream data model for applications that rely on rich connectivity – recommendation engines, knowledge bases, supply chains, life sciences, mobility, and more. Yet current graph systems exhibit a persistent gap relative to mature relational technology: they are strong at traversal, but weak at combined principled schema and temporality. Temporal extensions of relational systems are well understood, but contemporary graph databases rarely offer both sound typing and native multidimensional temporal history in one coherent model.

The query languages that accompany these systems inherit the same limitations. Most emphasise declarative pattern matching over loosely structured property bags and are only weakly or dynamically typed. Invalid traversals and misspecified property predicates are discovered late, at runtime, rather than prevented by the type system. Where temporal logic appears at all, it is typically pushed into ad hoc filters and application code, complicating optimisation and making correctness properties hard to state and enforce.

This work addresses these shortcomings on top of HASH (Chapter 4). We propose a language and compiler that combine traversal semantics with a functional calculus, native bi-temporality and a sound static type system informed by HASH's ontology. The language treats ontology (types) and knowledge (instances) uniformly as graph entities, supports links as first-class citizens (including links-to-links, i.e., T-junctions), and unifies traversals across ontology and knowledge.

A central design choice is to expose a traversal-oriented core rather than a purely declarative pattern language. Temporal axes enlarge the search space: what constitutes a "match" depends not only on topology and types but also on time. A traversal core, enriched with type-aware predicates and functional composition, makes these dependencies explicit and creates opportunities for cost-based planning, short-circuiting, and incremental evaluation.

The language core is syntax-independent. Multiple surface syntaxes can target a single authoritative AST, preserving one set of semantics. The compiler targets a heterogeneous execution environment: a scheduler with a cost model can place work across backends (virtual machine, database engines, permission policy evaluators, distributed mining), parallelise independent computations, and pipeline results across targets.

2. Requirements

To implement a query language atop HASH (Chapter 4), the language defined in this work must satisfy the following requirements.

HASH graph model. The language targets a labelled property graph (LPG) compatible model and operates over HASH's type system. Vertices represent either ontology or knowledge; edges may connect ontology to ontology, knowledge to knowledge, and knowledge to ontology. Properties are JSON-shaped and may be nested. The language must support uniform traversal of entities, links, and their types – including traversals between knowledge and ontology – and permit type-aware predicates.

Temporality. Native bi-temporal semantics (decision time and transaction time). The language must express as-of queries, bounded and unbounded intervals, overlap predicates, and coalescing. The design admits extension to a third temporal axis.

Traversal semantics. Because temporal axes enlarge the search space, the core must expose traversal semantics rather than a purely declarative pattern language.

Syntax-independent. The core is independent of surface syntax and admits multiple frontends. All frontends target a single authoritative AST.

Statically typed. The language exploits HASH's type system. Typing is sound and conservative: programs with statically detectable mismatches are rejected. The type discipline must model both ontology types and ordinary program types; local type inference is required to keep annotations lightweight. During analysis, the compiler must resolve any ontology types referenced by the query and use their declared structure to refine the static type during type checking.

Links as entities. Links are first-class entities (Chapter 4). The language must traverse links like any entity, support links-to-links (T-junctions), and handle multigraphs without special cases.

Heterogeneous execution. The design admits a heterogeneous computing environment. The compiler can target different backends (e.g., VM, database, permission policy engine, distributed mining). A scheduler with a cost model chooses placement (local vs. global), parallelises independent work, and enables pipelining across targets.

CRUD semantics. Beyond reads, the language must support create/update/delete of graph data with transactional semantics.

3. Related Works

Graph databases are commonly grouped into two families: resource description framework (RDF) and LPGs. They differ primarily in how they represent properties, identity, and schema.

3.1. Resource Description Framework

RDF [44] is a family of World Wide Web Consortium (W3C) standards for modelling and exchanging graph data. An RDF graph is a set of triples *(subject, predicate, object)*. Subjects are Internationalised Resource Identifiers (IRIs) or blank nodes; predicates are IRIs; objects are IRIs, blank nodes, or literals. IRIs are a superset of Uniform Resource Identifiers (URIs) and provide globally scoped identifiers with Unicode support; dereferencing is optional – an IRI need not resolve to a retrievable resource [9, 21]. By contrast, ontology identifiers in HASH are versioned URIs that must resolve (Chapter 4). Blank nodes denote existential resources without global identity (scope is limited to a graph/serialisation). Literals pair a Unicode lexical form with a datatype IRI (e.g. `xsd:dateTime`) or, for strings, a language tag [44].

RDF/1.1 is intentionally modular: the core spec defines the data model; companion standards provide schema, reasoning, constraints, and querying. Notable companions include RDF Schema (typing and simple vocabulary constructs) [19]; OWL (richer ontological modelling and description-logic entailment, geared towards the web) [56]; SHACL (constraint validation over RDF graphs) [27]; and SPARQL (declarative query and update language) [51]. Several serialisations are standardised – Turtle, TriG, and JSON-LD – reflecting RDF's serialisation-agnostic stance [16, 20, 31]. RDF semantics are open-world and monotonic, with entailment regimes (simple/RDF/RDFS/D) that allow engines to answer queries over the logical closure of a graph [44].

While this modular organisation keeps the core small and stable, it also pushes essential capabilities – typing, inference, and constraint checking – into optional components (RDFS, OWL, SHACL). In practice, deployments enable different subsets, so a graph that validates under one configuration may not under another, which complicates portability and reproducibility. Put differently, the core data model does not guarantee schema-level reasoning and constraint validation; these hold only when specific profiles are enabled [19, 27, 44, 56].

SPARQL is the primary query language for RDF. While initially considered, we found SPARQL to be too tightly coupled to the triple model and entailment assumptions, which differ significantly from HASH's typed LPG model. Additionally, SPARQL lacks standard bi-temporal operators; temporal extensions exist in the literature, but no widely adopted, standardised temporal RDF/SPARQL stack has emerged [63]. Although mappings between LPG and RDF are possible, they introduce non-trivial overhead and would obscure the guarantees and expressiveness HASH relies on [10]. Consequently, we do not pursue SPARQL further in this work.

3.2. Labelled Property Graphs

Labelled property graphs (LPGs) model graphs as vertices (nodes) and directed edges, where both vertices and edges may carry labels and key–value properties [7]. Labels group elements by role; properties typically range over scalar values, with some systems permitting nested collections [6, 10]. Unlike RDF, there is no standardised, globally scoped identifier scheme for labels or keys; with identity usually being graph-local.

Feature sets vary substantially across implementations [10]. This diversity shaped standardisation: ISO GQL [4] consolidates declarative pattern matching for property graphs, while SQL/PGQ integrates property-graph patterns into SQL for hybrid relational–graph systems [3]. To accommodate existing systems, these standards define a comprehensive core with optional features and conformance levels.

Schemas are often optional in the LPG family. A common practice is schema-on-read: applications attach meaning to labels and property keys without a globally enforced type discipline [6, 10]. Many products expose partial schema facilities (e.g., indexes, existence/uniqueness constraints), but the model does not assume a uniform, statically enforced schema layer [10].

Two query styles are prevalent. Declarative languages (Cypher/openCypher → GQL) express patterns to be matched [4]; evaluating such patterns reduces to subgraph search, whose core decision problems (isomorphism/homomorphism) are NP-complete in the worst case [17, 26]. Traversal languages (Gremlin) compose step functions from starting vertices; costs tend to scale with path length, yielding more predictable performance characteristics [45].

Temporality is not part of the core LPG model or mainstream query languages. Several research systems add temporal axes, with most focusing on uni-temporal graphs [15, 28, 33, 43]. A notable proposal, MAGMA, targets bi-temporal querying, but as of 2025 no public implementation or follow-up has been published [50].

Given that HASH adopts an LPG compatible model (Chapter 4), we evaluate both declarative (Chapter 3.3) and traversal (Chapter 3.4) approaches as potential bases for a typed, bi-temporal language before motivating a dedicated design.

3.3. Explored Avenue: Declarative Language

We initially considered extending an existing declarative language - most naturally, the newly published GQL standard - to support strongly typed, bi-temporal graph queries. However, every path explored required significant departures from the standard, negating the benefit of basing the implementation on an existing language. Extending openCypher was likewise impractical because the community is already moving towards GQL. OpenCypher no longer allows contributions to the specifications outside of aligning it more closely with GQL, making a forked dialect infeasible [1, 24].

The main blockers identified were:

- Implementation effort required for writing a compliant parser
- Differences between the HASH type system and the GQL type system
- Mismatch between the edge models
- The cognitive burden of expressing multi-temporal queries in a syntax not designed for them

3.3.1. Parser Implementation Effort

We found that there were no existing production-grade parser implementations for GQL in Rust, which meant that one would need to be implemented from scratch. Due to the sheer complexity of the language, which requires a specification of 610 pages, it was deemed out of scope for this project.

3.3.2. Type System Mismatch

Like early property-graph databases, Neo4j and Apache TinkerPop expose no fixed, typed schema: nodes are merely collections of labels and key–value attributes. The published GQL standard follows that model but offers an optional graph-type layer [4:4.13], which is defined as a set of:

- Node types - each with zero or more labels, and a set of zero or more property types.
- Edge types - each with zero or more labels, a set of zero or more property types, two endpoint node types, and a flag indicating whether the edge is directed.

Property types are simple pairs of identifiers and value types [4:4.16].

Although the schema facilities narrow the gap between GQL and the requirements of the existing type system, it still falls short due to:

1. **Semantic identifiers**: In HASH, a property name is part of the type's meaning; GQL treats names as opaque strings or aliases to types.
2. **Union support**: Unions are supported in GQL [4:4.14], but are ignored by most value expressions, whereas the existing type system relies on unions to mirror the flexibility of JSON schemas
3. **Recursive types**: Not supported in GQL
4. **Higher-order constructs**: Tuples, dictionaries/maps and closures are absent; only records [4:4.15.3] and lists [4:4.15.4] are available.

5. **Temporal primitives**: GQL only provides support for temporal instants and durations [4:4.16.6]; these primitives are insufficient in modelling bi-temporal queries, which rely on temporal ranges.
6. **Nullability semantics**: GQL, like SQL, treats `NULL` as an immaterial value [4:4.16.8]. This is in direct conflict with the HASH type system, which must explicitly distinguish between `nulls` and omitted values to align with JSON Schema's `null` and "optional" semantics.
7. **Types as first-class values**: The HASH type system requires that the data (entities) and their types live inside the same universe, allowing traversal between entities and their types. GQL separates graph data and types, and querying type information is unsupported [4:4.2.5].
8. **Inheritance**: The HASH type system allows for inheritance among types, whereas GQL has no concept of inheritance for user-defined types[1].

These gaps would necessitate extensive extensions or runtime checks, rendering GQL unsuitable as a foundation for our proposed system [4].

3.3.3. Edge Model Incompatibility

Traditional property graphs, such as GQL, treat edges as a separate concept from nodes. By contrast, the HASH graph does not distinguish between nodes and edges. Instead, edges - called links - are instances of entity types which inherit from the `https://blockprotocol.org/@blockprotocol/types/entity-type/link/v/1` type. Each link has mandatory link data attached, which consists of the left and right entities it points to. Having these higher-order edges enables some interesting use cases and possibilities, such as links pointing to other links.

GQL cannot model these higher-order edges directly. The closest approximation is to normalise these links by decomposing these edges and directly exposing the left and right edges. Allowing for queries similar to:

```GraphQL
MATCH
  (:Entity {id: L})
  <-[:LEFT]-(:Entity:Link {id: X})-[:RIGHT]->
  (:Entity {id: R})
```

Listing 1: GQL Higher-Order Edge Query

This normalisation introduces several drawbacks:

- **Weaker invariants**: The graph guarantees that there is always exactly one pair of left-right edges per link entity. GQL is unable to encode this guarantee, resulting in a significant subset of queries that are syntactically valid in GQL but which would always return an empty set.
- **Pattern expansion**: Queries that once matched a single link must now always match a pattern, which would incur significant overhead. One can partially mitigate this issue by specialising these patterns in the compiler, but this in itself would be a

[1] Inheritance could be emulated, by requiring that each type being inherited from is present as a label and in turn each inherited from type has a node type definition

challenging task due to the different permutations possible, such as directionality, labels or aliases.
- **Standards-compliance overhead**: To remain GQL-compatible, one would need to expose and implement a significant amount of functionality that is superfluous for our use case, such as variable-length path syntax, per-edge predicates, and DDL for standalone edge types, among others. Much of the engineering work would be for nought.

Because HASH's link-as-node semantics conflict with GQL's edge-primitive design, adopting the standard would mean extra engineering work for unused features and less ergonomic queries: patterns that are a single hop in the HASH graph would become multi-step constructs in GQL, increasing the risk of hard-to-debug queries, which are syntactically valid but semantically incorrect [4].

3.3.4. Cognitive Overhead

Adopting GQL is technically possible, but only by significantly diverging from existing specifications, thereby defeating the initial goal of adopting said existing specification (see Chapter 3.3.2 and Chapter 3.3.3).

Appendix A demonstrates that phrasing a simple bi-temporal predicate in GQL inflates a one-hop pattern into multiple joins and explicit interval guards, increasing both syntax and cognitive load. A language with native temporal operators expresses the same intent as a single hop.

3.3.5. Compiler Implementation Effort

Implementing a compiler that translates a declarative graph query into an executable plan is a non-trivial task. The compiler must first parse the patterns and build an operator graph, and linearise it into a pipeline. It must then choose an evaluation order that minimises intermediate results, which is typically tackled through the use of heuristics. Because these optimisation decisions are opaque to the user, minor syntactic edits can result in radically different query plans and unpredictable runtimes.

The core pattern matching of a declarative query relies on subgraph search; the engine looks for subgraphs that, depending on the implemented semantics, are either homo- or isomorphic to the query pattern provided. Subgraph isomorphism is NP-complete [17], and homomorphism is no easier in the worst case [26]. Bi-temporal queries significantly enlarge the search space by adding two interval-overlap dimensions, thereby increasing the number of candidate matches [7].

Our architecture introduces a second layer of complexity: The compiler must split the query plan across multiple execution contexts, such as a VM, the underlying PostgreSQL database, and the policy engine, with additional targets planned for the future. The planner would therefore be required to annotate each operator with a target context and insert the requisite data transfer steps. Implementing that cross-context planner for the full GQL dialect would dwarf the effort of developing our smaller, purpose-built language.

3.4. Explored Avenue: Functional Language

We next considered a functional traversal language. The best-known example is the non-standardised Gremlin language from Apache TinkerPop, where a query specifies a start vertex and a sequence of traversal steps. This approach avoids the NP-complete subgraph-isomorphism [17] problem by relying instead on path traversal, whose cost is roughly linear in path length, resulting in more predictable runtimes. However, users may find that the resulting query is more complicated to read than its declarative equivalent.

Gremlin, furthermore, is tightly bound to Groovy and the TinkerPop stack, and therefore the Java Virtual Machine (JVM); adopting it would entail using only the Groovy-like syntax and completely abandoning the existing stack, as we would require implementing a heterogenous computing environment, thereby nullifying much of the benefit of using an existing language or framework, especially since our existing PostgreSQL based graph implementation is written in Rust.

Moreover, Gremlin offers neither typed schemas nor native bi-temporal operators. Prior studies reinforce these gaps Jaewook Byun, Sungpil Woo, and Daeyoung Kim. [15] successfully extend Gremlin, adding a uni-temporal axis, indicating that the approach of adding bi-temporal support to a functional traversal language is feasible, although as Shriram Ramesh, Animesh Baranawal, and Yogesh Simmhan. [43] note how Chrono-Graph has several inherent limitations, in that it does not support distributed execution and is lacking any temporal operators or data types, and therefore, making operations involving time manipulation impossible. Additionally, Norbert Tausch, Michael Philippsen, and Josef Adersberger. [53] note how Gremlin is fundamentally unsuited for use in a statically typed context due to its strong relation to Groovy, which runs counter to our stated design goals.

Because a Gremlin fork cannot deliver static typing, nor can it operate in a bi-temporal or heterogeneous computing environment, we decided instead to design a new, typed, bi-temporal functional traversal language rather than implement full GQL or adapt Gremlin.

4. HASH

HASH [55] is an open-source, AGPL-licensed, strongly typed, bi-temporal graph database and type system.

The type system is a superset of JSON Schema [58] and can be expressed as a set of validation schemas. All values are therefore JSON-shaped; the usual caveat applies: binary payloads cannot be represented natively. The type system distinguishes between ontology (the shapes and their semantics) and knowledge (the data). Knowledge consists of entities, whereas ontology consists of:

- Data types – these describe value domains, which can be optionally constrained and grouped (e.g. a `Currency` group with `Euro`, `U.S. Dollar` and `Yen`).
- Property types – give semantic meaning and structure to a value, either by referencing data types or by forming objects whose fields are other property types, potentially creating cyclic structures.
- Entity types – specify entity schemas, allowed outgoing links, and inheritance.

These ontology kinds compose naturally: data types may be grouped; property types may reference other property types (as object fields) or data types; entity types are assembled from property types and may declare both inheritance and link constraints. Multiplicity is explicit: a property may be singular or an array at any nesting depth. With multiple inheritance, conflicting multiplicities for the same property make the entity type invalid; such types are rejected at ontology creation.

Ontology items have global identity and documented semantics. Each definition carries a name, a description, a version, and a URL identifier. The system is decentralised (in contrast to central registries such as schema.org [2]). To avoid name clashes across independently published vocabularies, object properties refer to their property types by URL: the property key is the URL of its property type. This yields collision-free composition at the cost of longer keys.

Edges are not a separate primitive. In HASH, links are ordinary entities: any entity type that extends `https://blockprotocol.org/@blockprotocol/types/entity-type/link/v/1` denotes a relationship between two entities. Because links are entities, they can carry properties. They can themselves be the target of other links (links-to-links, enabling T-junctions), which makes patterns such as hypergraphs straightforward to represent.

To ensure a consistent graph at all times, validation occurs on every modification: entity writes must satisfy their entity-type schemas and updates to the ontology vocabulary must be internally consistent (including inheritance and multiplicity checks). The graph

is bi-temporal: both data and types can be queried as of valid time and transaction time. Because ontology lives in the same graph as data, queries can introspect types alongside entities – for example, they can constrain results to entities that instantiate a particular type or expose a given property type via inheritance. This integration allows us to improve type safety at query time without sacrificing usability.

5. Language Design

HashQL is a statically typed functional calculus with traversal semantics. Its core is pure and referentially transparent: every expression is a value or a composition of values, and no construct can read or mutate external state.

All interaction with the bi-temporal graph is isolated by the effect type `Graph<T>` [37, 57]. Computations inside the graph, be it read or write, may only happen inside that type. `Graph<T>` values are first-class and composable, but **opaque**: they can be executed only by the host runtime at the program boundary. This separation follows the standard monadic pattern for effect isolation [37, 57]. Consequently, ordinary expressions remain side-effect free, and equational reasoning is preserved for all user code that does not explicitly handle the `Graph<T>` effect.

5.1. Frontend Syntax

The compiler is built around a syntax-independent core, akin to LLVM [30], where all the different compilation stages, such as abstract syntax trees or high-level intermediate representations, remain unbound from any surface syntax. That separation incurs some overhead during implementation, but allows us to develop richer front-ends later, without disrupting the existing ones. For example, a future declarative layer could lower GQL-style homomorphism patterns to functional calls, or an imperative layer might adopt a Rust-like surface syntax.

For the initial front-end, we prioritised:

1. Ease of parsing
2. Ease of machine generation
3. Sufficient expressiveness to cover all language constructs

The ability to machine-generate programs outweighs the convenience of hand-authoring, as most queries will be produced by host-language domain-specific languages, such as Rust, TypeScript, or similar languages, that generate queries programmatically. A machine-friendly grammar makes creating these domain-specific languages (DSLs) concise, type-safe and straightforward to generate.

The chosen syntax is *J-Expr*, a JSON-with-comments (JSONC, see: Appendix C) analogue of S-Expressions. Most languages already provide mature JSON (de)serialisation libraries, and the S-Expressions-inspired shape should feel immediately familiar to developers versed in functional languages such as Lisp.

J-Expr comprises just four forms:

- **Underscore** – _
- **Path Expression** – symbolic path
- **Call Expression** – function call
- **Data Expression** – literal or composite value

An approximate JSON schema for the grammar is available in the appendix (see Appendix B).

5.1.1. Underscore

The underscore, represented by the JSON string `"_"`, denotes either an expression whose value is discarded or an inferred type. It corresponds to the underscore node in the abstract syntax tree.

5.1.2. Path Expression

Paths are represented as strings in J-Expr. Unlike other parts of the grammar, which host-language DSLs will generate, we expect that paths and their components will be used directly by human authors, irrespective of any machine-generated code. To make these expressions easier to write for human authors, especially in the present situation, where no host language DSL exists, we have devised a small embedded DSL.

The DSL covers three components:

- Access parsing
- Path parsing
- Type parsing

All parts of this embedded grammar, except for the parsing of paths, can be written in plain J-Expr and do not need the use of an embedded DSL. The DSL exists to supplement these constructs, making human authoring easier, but not to replace any existing syntax.

Access parsing is only available at the top level and translates:

- `<expr>.<ident>` ⟶ field access
- `<expr>[<expr> | <int>]` ⟶ index access

We deliberately disallow arbitrary J-Expr inside the brackets when accessing indices. Allowing it would bloat the grammar and would undermine the goal of J-Expr of remaining primarily machine-generable. Initial experiments show that index access almost always either refers to a constant integer or a bound variable, which the restricted form supports and expands upon.

```JSON
"::core::ops::lte"
"::core::math::add<Number, Integer>" // generic arguments
```

Listing 2: J-Expr Absolute Paths

```json
"foo"
"ops::lte"
"Foo<T, U: Integer | String>" // mix of generic arguments and constraints
```

Listing 3: J-Expr Relative Paths

```json
"a.b" // is the same as:
[".", "a", "b"]

"a[b]" // is the same as:
["[]", "a", "b"]

"a.b[2].c[d]" // is the same as:
["[]", [".", ["[]", [".", "a", "b"], {"#literal": 2}], "c"], "d"]
```

Listing 4: J-Expr Embedded DSL Access

The complete grammar is available in the appendix (see Appendix B).

5.1.3. Call Expression

Call expressions are central to HashQL, and therefore to J-Expr. They are written as JSON arrays of expressions, where the first element is the callee, and the remaining elements are its arguments. Arguments may be unlabelled or labelled. Although HashQL itself has planned for labelled arguments, it only has initial support in the AST.

A labelled argument is an object whose single key begins with a colon (`:`). The colon distinguishes labels both from regular paths as well as other constructs of the language, as a colon (`:`) is invalid outside of labelled arguments. HashQL follows Swift's convention: arguments remain positional but may carry an auxiliary label, so each object must contain exactly one labelled argument. Allowing multiple keys would clash with JSON's semantics, which state that objects should be unordered [12].

For convenience, an identifier that starts with a colon on its own (`":name"`) is expanded to the equivalent single-entry object of `{":name": "name"}`.

```json
// Call function add with variables a and b
["::core::math::add", "a", "b"]

["foo", {":a": "a"}, {":b": "b"}] // is equivalent to:
["foo", ":a", ":b"]
```

Listing 5: J-Expr Call

5.1.4. Data Expression

Data expressions construct the elementary data types in HashQL. They are written as a JSON object that contains one primary key-value pair and may include additional optional entries.

Every key in such an object must begin with an octothorpe (#). We chose # because it is not part of any valid identifier (see Figure 18) and does not clash with the leading colon (:) that marks labelled arguments (Chapter 5.1.3).

The recognised keys are:

- `#literal` – JSON primitive literal
- `#struct` – closed struct
- `#struct(open)` – open struct
- `#tuple` – tuple
- `#dict` – key-value dictionary
- `#list` – list

Any of these objects may include an additional `#type` entry that refines the value's type. The value of the `#type` entry must be a string that follows the embedded DSL rule `type` in Listing 49. During desugaring, `#type` expands to a call to the special-form `::kernel::special_form::as` function. This provides human authors with the ability to use a more concise notation when explicitly specifying the type of an expression.

Depending on context, a data expression is interpreted as either a value or a type. It is treated as a type only when it appears in designated argument positions of kernel special-form calls. User-defined functions always receive concrete values and cannot directly receive types as values.

```JSON
// Both expressions are equivalent:
{
  "#struct": { "a": "a", "b": "b" },
  "#type": "(a: Integer, b: _)"
}

["::kernel::special_form::as",
  { "#struct": { "a": "a", "b": "b" } },
  { "#struct": { "a": "Integer", "b": "_" } }
]
```

Listing 6: J-Expr Type Desugaring

5.1.4.1. `#literal` Expression

The `#literal` expression introduces a primitive literal. It covers all primitives defined in the HashQL Type System: Integer, Number, String, Null, and Boolean.

In J-Expr, a literal is written as a JSON scalar. Numeric scalars, as described in Tim Bray. [12], are classified in J-Expr depending on whether they have an exponent or a fractional part. A numeric scalar without either is parsed as an Integer; otherwise, it is parsed as a Number.

```json
{"#literal": "value"} // type: String
{"#literal": 123}     // type: Integer
{"#literal": 123.45}  // type: Number
{"#literal": true}    // type: Boolean
{"#literal": null}    // type: Null
```

Listing 7: J-Expr Literal Expression

5.1.4.2. `#struct` and `#struct(open)` Expression

The `#struct` expression defines a struct – a heterogeneous mapping from identifiers to values. Structs come in two variants: closed structs are invariant in their set of fields, whereas open structs are covariant in that regard. A closed struct can be safely coerced to an open struct, but not vice versa (S-STRC2).

In J-Expr, a struct is written as a JSON object, where every key must be unique and whose keys must be valid HashQL identifiers (see Figure 18), and their value may be any valid J-Expr.

Within the embedded DSL, open structs are differentiated from closed ones by a leading octothorpe (`#`). We chose to add additional explicit syntax to open structs, rather than closed structs, as we expect the latter to be more common.

```json
// Closed struct:
{ "#struct": { "a": { "#literal": 2 }, "b": { "#literal": 3 } } }

// Open struct:
{ "#struct(open)": { "a": { "#literal": 2 }, "b": { "#literal": 3 } } }
```

Listing 8: J-Expr Struct Expression

5.1.4.3. `#tuple` Expressions

The `#tuple` expression defines a tuple – an ordered, fixed-size collection of heterogeneous elements that is invariant in length.

In J-Expr, a tuple is written as a JSON array, where each element may be any valid J-Expr.

```json
{"#tuple": ["a", {"#literal": 2}, "c"]}
```

Listing 9: J-Expr Tuple Expression

5.1.4.4. `#dict` Expression

The `#dict` expression defines a dictionary – an unordered, dynamically sized collection of homogeneous key-value pairs. Its type is `::kernel::type::Dict<K, V>`, where `K` is an invariant key type and `V` a covariant value type.

Because dictionaries allow for arbitrary keys and values, J-Expr offers two concrete syntaxes to define them. When keys are simple strings, as is the case in the majority of instances, a JSON object is used to make authoring more convenient for humans. Unlike structs (see Chapter 5.1.4.2), where the keys are interpreted as paths (see Chapter 5.1.2), keys are treated as literal strings. For non-string keys, a two-dimensional array is used, where each element of the outer array describes an entry. Each entry is a

two-element array, with the first element being the key and the second element being the value. Both key and value may be any valid J-Expr.

```JSON
// Both expressions are equivalent:
{ "#dict": { "foo": { "#literal": "bar" }, "baz": "alice" } }

{ "#dict": [
  [ { "#literal": "foo" }, { "#literal": "bar" } ],
  [ { "#literal": "baz" }, "alice" ]
] }
```

Listing 10: J-Expr Dictionary Expression

5.1.4.5. `#list` Expression

The `#list` expression defines a list – an ordered, dynamically sized collection of homogeneous values. Its type is `::kernel::type::List<T>`, where `T` is a covariant value type.

In J-Expr, a list is written as a JSON array, where each element may be any valid J-Expr.

```JSON
{"#list": [{"#literal": 1}, {"#literal": 2}, "c"]}
```

Listing 11: J-Expr List Expression

5.2. In-Program Execution of Graph Effects

Currently, the language does not allow the evaluation of `Graph<T>` effects inside user programs. Still, the safe evaluation of `Graph<T>` concerning referential transparency can be guaranteed, given some additional invariants, as every `Graph<T>` is tied to a bi-temporal slice, where one axis is a single instant, referred to as being pinned. The other axis is a half-open interval $[\text{start}, \text{end})$, referred to as being variable. The user cannot specify the same axis to be pinned and variable; each query must specify one axis as pinned and one as variable. Additionally, each entity in the graph has a recorded instant for each axis.

Denote:

$$
\begin{aligned}
\tau &\coloneqq \text{transaction time argument (instant or interval)} \\
\delta &\coloneqq \text{decision time argument (instant or interval)} \\
\tau_0 &\coloneqq \text{real clock time at program start (instant)} \\
\tau_{\max} &\coloneqq \begin{cases} \tau_{\text{end}} & \text{if } \tau \coloneqq [\tau_{\text{start}}, \tau_{\text{end}}) \\ \tau & \text{otherwise} \end{cases}
\end{aligned}
\tag{1}
$$

Figure 1: Definition of Time Axes

Referential transparency holds **iff** $\tau_{\max} \leq \tau_0$. In words: the largest transaction time named in the query must not exceed the program's start time.

The store is append-only; therefore, any update executed during the run commits with $\tau' > \tau_0$. Such entities are always outside the specified bi-temporal slice of any query

and are consequently invisible. For the evaluation of a graph query to be referentially transparent, the runtime must reject any query which specifies $\tau_{\max} > \tau_0$ or defer its execution to the program boundary, since such a query could observe future writes.

Decision time needs no separate check. Even if a newly created entity is back-dated with a decision time that falls inside the query's $[\delta_{\text{start}}, \delta_{\text{end}})$ interval, its transaction time is necessarily the moment of insertion and therefore $\tau' > \tau_0$; since τ' lies outside the slices guarded by $\tau_{\max} \leq \tau_0$, the entity remains invisible and referential transparency is preserved.

Under these constraints, pure operations, such as building graphs from entity lists, merging graphs, or returning newly created graphs from update operations, remain fully referentially transparent.

5.3. Namespacing

In HashQL, a universe – also referred to as a namespace in other languages, or informally a realm – is a logical grouping of declared names. Names are separated into distinct universes according to the kind of entity they denote, meaning that the same identifier may refer to different entities without conflict.

HashQL defines two universes:

- **Value universe** – run-time constructs: literals, composite values, and callable terms (functions).
- **Type universe** – compile-time constructs: type aliases, type constructors, and constraints consumed only by the type checker.

This strict split mirrors Rust's namespacing rules, keeping name resolution predictable and enabling separate reasoning about the types of a program and its actual execution [49:12.1]. After type checking, HashQL erases all type metadata, retaining only value-universe constructs.

During the design phase, we surveyed various strategies for namespacing employed by statically typed languages.

Separate universes: Languages like Rust keep universes completely disjoint, where an entity is unable to cross over to a different universe. Unlike HashQL, Rust distinguishes between five different namespaces: type, value, macro, lifetime, and label. This simplifies name resolution, making it more predictable and allowing for exclusive type checking during compilation, which makes the types essentially zero-cost at runtime [49:12.1].

Runtime reflection is an extension of separate universes, in which the runtime is allowed to access type information. Reflection allows for the introspection of types, but not their modification, enabling a one-directional flow of information and weakening the clear separation between the two. A prominent example is Java, which has extensive reflection capabilities that are used by frameworks and libraries such as Jackson and Spring for dependency injection and automatic configuration [38, 62]. Another prominent example is C++, which will gain reflection capabilities in the C++26 edition [59].

Rust has no official support for reflection[2]. Still, user-provided macros, such as facet, allow for an approximation in user code [5].

Types as values collapses the distinction at compile-time, allowing for the construction and handling of types as if they were ordinary values. Zig, an ahead-of-time (AOT) compiled language, utilises types as values and completely forgoes the concept of generics. Instead, it allows for the construction of types through functions, as long as the expression can be evaluated at compile time [64:Generic Data Structures].

Refinement types enrich existing types with predicates evaluated at compile time. A prominent example is LiquidHaskell, in which one can state that, e.g. an `Int` must always be positive [46, 54].

Conditional types add a type-level if. The compiler tests a candidate type against the condition; if it matches, the result is the then branch, otherwise the else branch applies. Languages that need to encode dynamic patterns at compile time – TypeScript is the canonical example – depend on this feature. The added expressiveness raises both implementation effort and type-checking time [35:Conditional Types].

Dependent types allow arbitrary terms in type positions. Proof-oriented languages, such as Coq and Idris, adopt this model, gaining extreme expressiveness at the cost of a substantially more complex implementation and slower compilation times [11, 60].

Many of these approaches are orthogonal, and a language may implement multiple of these.

HashQL adopts the first strategy of separate universes, as it provides a clear mental model, has been proven in existing systems, is straightforward to implement, and allows for a clear separation of execution modes. Future iterations of the language may extend the language to feature refinement types, as they are required to accurately model the data types of the HASH Type System, which rely on JSON schema constraints.

```JSON
// Type universe: alias Id to Integer
["type", "Id", "Integer",
// Value universe: bind a constant with the same name
["let", "Id", "Id", {"#literal": 42},
// In 'let x: Id = Id', the annotation `Id` is the type; the expression `Id` is the value
["let", "x", "Id", "Id",
"x"
]]]
```

Listing 12: Namespacing (J-Expr)

[2]Exploration through the use of Rust Foundation Grants into adding reflection to the core language exists, most notably [29]

5.4. Expression Forms

HashQL is an expression language, meaning that every syntactic construct evaluates to a value and may appear wherever a value is expected. The language comprises four kinds of expressions: paths, calls, data literals and modules.

5.4.1. Path

A path resolves to a previously declared value, which may be defined in another module. A path cannot cross over universes: a path declared in the value universe can only reference values, and a path in the type universe can reference only types. They also cannot target a module itself; a module must first be imported via `::kernel::special_form::use`, after which its contents can be accessed using relative paths (see Chapter 5.5, Chapter 5.6).

5.4.2. Call

A call applies a closure value to its positional – optionally labelled – arguments and returns the result of the closure invocation. Evaluation order is strict; all arguments are evaluated before the call is made.

Higher-level operations, such as defining new types or binding a specific expression to an identifier, are realised through special-form intrinsic closures that the compiler rewrites early on into dedicated nodes in the AST (see Chapter 5.5).

5.4.3. Data Literal

Data literals denote concrete immutable run-time values. They are modelled as a separate class of expressions, rather than being constructed through special-form closures, to maintain the consistency of the language's semantics. Constructing more complex data literals, such as structs or tuples, through special-form closures would require the introduction of variadic arguments, which are outside the scope of this project.

5.4.3.1. Primitive

The language defines five primitive atoms:

- **Integer**: arbitrary-precision integer
- **Number**: arbitrary-precision floating-point value
- **Boolean**: true or false
- **Null**: explicit absence of a value
- **String**: arbitrary-length ordered sequence of Unicode scalar values

5.4.3.2. Struct

A struct denotes a finite, fixed-field, unordered mapping $\sigma : K \to V$, where K is a finite set of field identifiers, and V the set of all values. Each key is a compile-time constant and must be a valid identifier (see Appendix B). The mapping is heterogeneous; unlike dictionaries, different fields may have unrelated value types.

A struct is *open* when it admits width subtyping: a struct with additional fields is a subtype of one without them, and is therefore covariant in its field set. A *closed* struct does

not admit width subtyping and is therefore width-invariant. In both cases, each field position remains covariant concerning its declared value type (see formal subtyping rules in Chapter 6).

5.4.3.3. Tuple

A tuple denotes a finite, fixed-length, ordered mapping $\tau : \{0, \ldots, n-1\} \to V$, where the index set $\{0, \ldots, n-1\}$ is fixed at compile time and captures the element order. The mapping is heterogeneous; unlike lists, different elements may have unrelated value types.

Each element is covariant in its declared value type, while the tuple length itself is invariant (see formal subtyping rules in Chapter 6).

5.4.3.4. Dictionary

A dictionary denotes a finite, dynamically sized, unordered mapping $\delta : K \to V$, where K is the set of all keys, and V is the set of all values. Keys may be computed dynamically at runtime.

It is homogeneous:

$$
\begin{aligned}
&T_K \coloneqq \text{declared key type of } \delta \\
&T_V \coloneqq \text{declared value type of } \delta \\
&\text{type}(x) \coloneqq \text{type of value } x \\
&\forall k \in \text{dom}(\delta).\text{type}(k) = T_K \\
&\forall v \in \text{ran}(\delta).\text{type}(v) \sqsubseteq T_V
\end{aligned}
\tag{2}
$$

Figure 2: Dictionary homogeneity constraints

Dictionaries are represented through the intrinsic type `::kernel::type::Dict<K, V>`, which is invariant in its key type `K` and covariant in its value type `V` (see formal subtyping rules in Chapter 6).

5.4.3.5. List

A list denotes a finite, dynamically sized, ordered mapping $\ell : \{0, \ldots, n-1\} \to V$, where the contiguous index set $\{0, \ldots, n-1\}$ captures element order, and V is the set of all values. Indices may be computed at runtime.

It is homogeneous:

$$
\begin{aligned}
&T_V \coloneqq \text{declared value type of } \ell \\
&\text{type}(x) \coloneqq \text{type of value } x \\
&\forall v \in \text{ran}(\ell).\text{type}(v) \sqsubseteq T_V
\end{aligned}
\tag{3}
$$

Figure 3: List homogeneity constraint

Lists are represented through the intrinsic type `::kernel::type::List<T>`, which is covariant in its value type `T` (see formal subtyping rules in Chapter 6).

5.4.4. Module

Modules package a set of bindings (values and types) into a single first-class value that can be imported and subsequently accessed. User code is currently unable to declare new modules directly. Chapter 5.6 details current restrictions and future extensions.

5.5. Special Forms

To express constructs beyond the base expressions defined in Chapter 5.4 – such as binding a value to an identifier (`let`), conditional evaluation (`if`), or module import (`use`) – HashQL employs special-form closures. These compiler-defined macros expand into dedicated nodes during AST lowering. The design is inspired by prior work in functional languages, such as Scheme, Racket and Elixir [23:Kernel.SpecialForms, 32:2, 34:3].

Special form closures differentiate themselves from regular intrinsics in timing: they are rewritten in the AST phase, whereas regular intrinsics specialise during HIR lowering.

All special forms are defined in the `::kernel::special_form` module and are auto-imported in the prelude. This enables modules to refer to the special forms as plain identifiers without needing an explicit import or to refer to them by their absolute path. Users may rebind or shadow them, effectively aliasing or disabling a special form, but are unable to define macros themselves. The potential for user-defined macros is discussed in Chapter 8.

Special-form closures obey two additional rules:

1. **Macro-phase restriction**: because a special form expands to a concrete AST node during lowering and does not exist as a runtime value, it cannot be passed as an argument when calling higher-order closures or be stored in data structures.
2. **Fixed arity and kinds**: A special form closure cannot be partially applied; the macro expands before type inference, so the compiler assumes the declared arity and argument kinds when it performs the rewrite.

```json
[
  // fn<T>(value: T): T -> value
  ["fn", {"#tuple": ["T"]}, {"#struct": {"value": "T"}}, "T", "value"],
  "let"
]

// Error: Special form cannot be used as a value
```

Listing 13: Passing a special form to a closure is a compile-time error

```
["let", "foo", "let"
["foo", "a", {"#literal": 2}
"a"
]]
// evaluates to: 2
```

JSON

Listing 14: Aliasing a special form closure and invoking it

```
["let", "let", {"#literal": 2},
"let"
]
// evaluates to: 2
```

JSON

Listing 15: Shadowing a special form closure binding

5.5.1. Notation

We write a special form's signature as `name/arity<generics>(args) -> return where-clause`, e.g.:

```
let/3<T, U>(name: Ident, value: Expr<T>, body: Expr<U>) -> U

as/2<T,U>(value: Expr<T>, type: Type<U>) -> Expr<U> where T <: U
```

Signature

Listing 16: Notational Example

An argument can have the following kinds:

- **`Ident`**: requires a valid identifier
- **`Ident<T>`**: requires an identifier that must be exactly `T`
- **`Path`**: requires a valid path
- **`GenericIdent`**: requires a valid identifier, with optional generic constraints
- **`Expr<T>`**: expression that evaluates to `T`
- **`Type<T>`**: type that is equivalent to `T`
- **`List<T>`**: requires a list of `T`
- **`Dict<K, V>`**: requires an unordered mapping of `K` to `V`
- **`Struct`**: requires a struct
- **`Tuple`**: requires a tuple
- **`Natural`**: literal integer value in $\mathbb{N}$ (including 0)
- **`T.field`**: only valid if `T` is a struct or tuple; type of the field/element specified by `field`

The *where-clause* is used to specify additional constraints that must hold. We define the following constraints:

- **`T <: U`**: `T` must be a subtype of `U`
- **`param ∈ T`**: closure parameter `param` must be a field inside the struct `T`
- **`|T|`**: total amount of fields of struct or tuple `T`
- **`n ≤ m`**: `n` must be less than or equal to `m`
- **`n < m`**: `n` must be strictly less than `m`

5.5.2. `if`

The `if` special form enables conditional evaluation based on a provided `test` predicate. It supports two different signatures:

```
1  if/2<T>(test: Expr<Boolean>, then: Expr<T>) -> Option<T>          Signature
2  if/3<T>(test: Expr<Boolean>, then: Expr<T>, else: Expr<T>) -> T
```

Listing 17: `if` special form signature

If no `else` branch is provided, the value is implicitly wrapped in `::core::option::Option<T>`, yielding `::core::option::None` when the predicate evaluates to `false`, and `::core::option::Some<T>` when it evaluates to `true`.

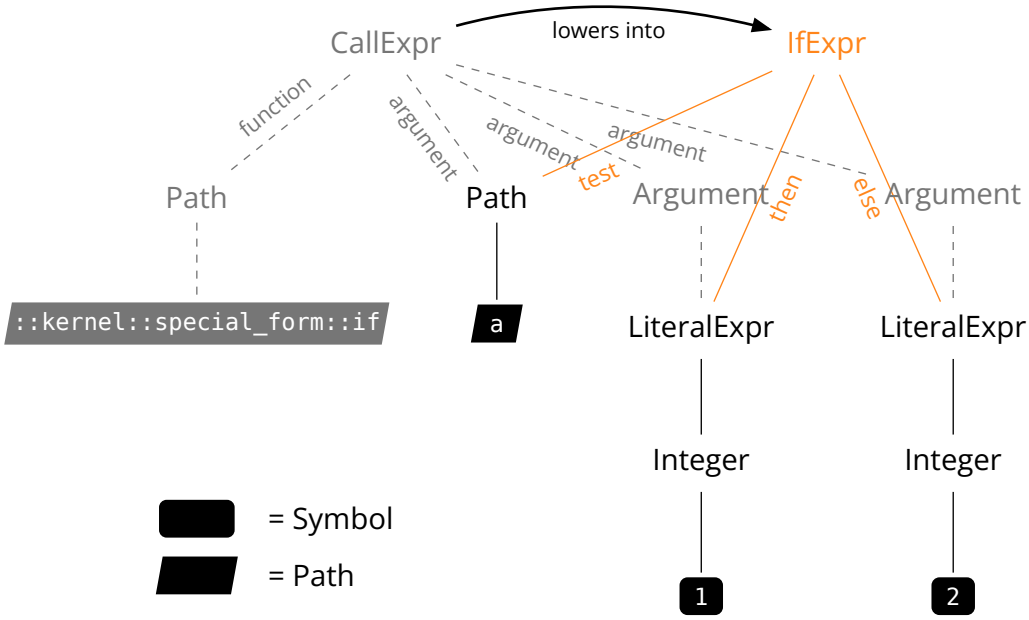


Figure 4: Lowering of `["if", "a", {"#literal": 1}, {"#literal": 2}]` in the AST

5.5.2.1. Compiler Status

`if` is rewritten and handled in the AST. The evaluator, which transforms the HIR into an equivalent PostgreSQL query, utilises the `SelectCompiler` – a component of the existing HASH Graph. This `SelectCompiler` currently has some fundamental limitations, one of which is the inability to represent conditional clauses. Therefore, during reification of the AST into the HIR, the compiler will emit a diagnostic for every `if` expression encountered and will subsequently abort compilation. Full support of conditional expressions will require an extension of the existing `SelectCompiler` and implementing a virtual machine (VM) for execution of arbitrary instructions as outlined in Chapter 8.

5.5.3. `as`

The `as` special form introduces an explicit up-cast (type assertion). The assertion is verified during HIR type checking and subsequently erased, meaning that they do not incur any run-time overhead.

```
1  as/2<T, U>(value: Expr<T>, type: Type<U>) -> Expr<U> where T <: U  Signature
```

Listing 18: `as` special form signature

The type of `value` must be a subtype of the specified type; hence `as` can only broaden a value's type (upcast). Attempts to narrow the type (downcast) are rejected because they would break type soundness. The HIR allows for forced type assertions, which are currently unused, but enable the compiler to generate unchecked type assertions. This mechanism is not exposed in the AST and, therefore, unavailable in user code.

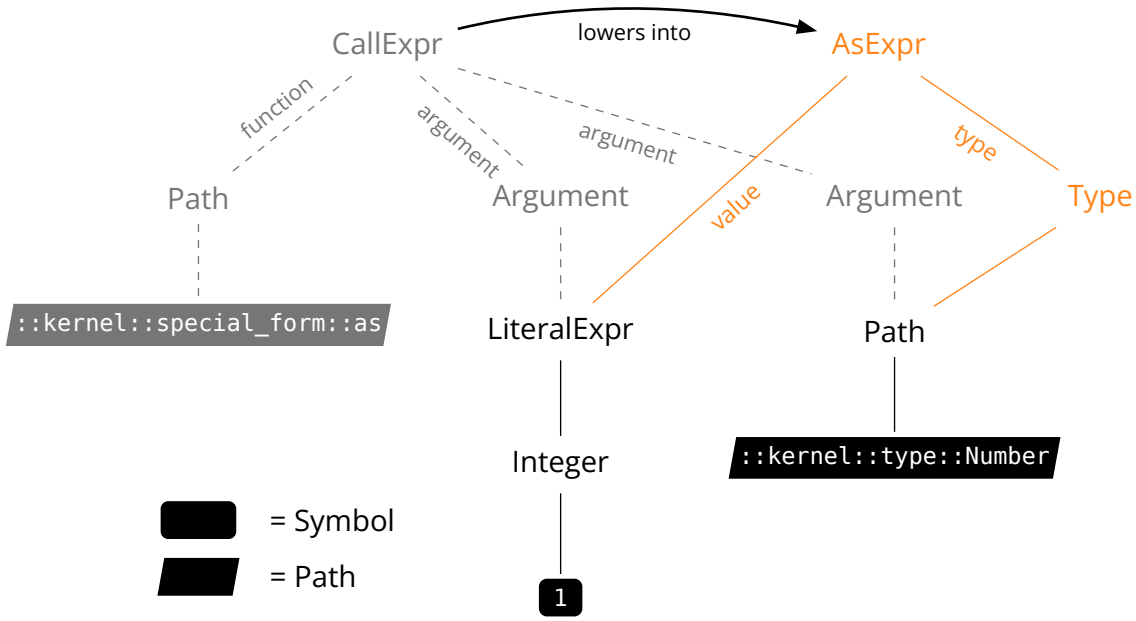


Figure 5: Lowering of `["as", {"#literal": 1}, "Number"]` in the AST

5.5.4. `let`

The `let` special form introduces a variable binding whose scope is limited to the given body expression. It optionally accepts an explicit type annotation for the bound value and has two signatures:

```
let/3<T, U>(name: Ident, value: Expr<T>, body: Expr<U>) -> Expr<U>
let/4<T, U, V>(name: Ident, type: Type<T>, value: Expr<U>, body: Expr<V>) -> Expr<V> where U <: T
```

Signature

Listing 19: `let` special form signature

The type annotation is syntactic sugar. During the reification from the AST into the HIR, the compiler desugars it into a call to the `as` special form –consequently, the same subtyping restriction described in Chapter 5.5.3 applies.

```
// explicit annotation:
["let", "foo", "Number", {"#literal": 2}, "foo"]
// desugared form
["let", "foo", ["as", {"#literal": 2}, "Number"], "foo"]
```

JSON

Listing 20: Annotated `let` and its desugared form

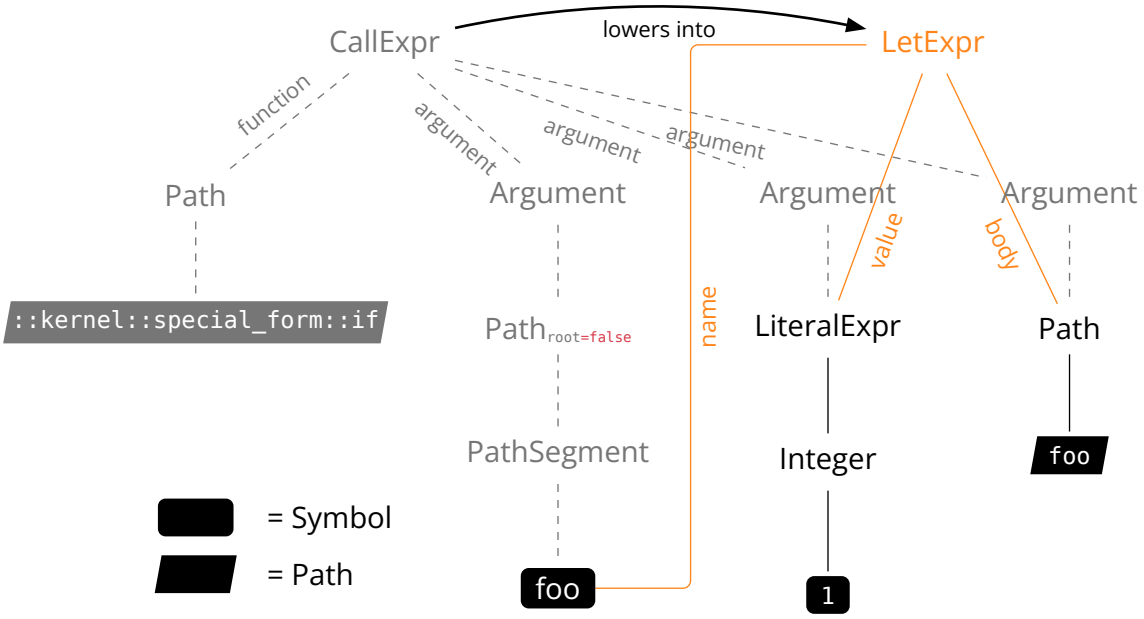


Figure 6: Lowering of `["let", "foo", {"#literal: 2}, "foo"]` in the AST

5.5.5. `type`

The `type` special form introduces a type alias whose scope is limited to the given body expression.

```
type/3<T, U>(                                                    Signature
  name: GenericIdent,
  value: Type<T>,
  body: Expr<U>
) -> Expr<U>
```

Listing 21: `type` special form signature

Unlike a `let` binder, the identifier is in scope inside the `value` argument to allow for self-referential and mutually recursive types. A `type` binder may also carry generic constraints, which a `let` binder would reject.

```
["type", "List<T: Number>",                                      JSON
    { "#struct": { "value": "T", "next": "Option<List<T>>" } },
{ "#literal": 2 }]
```

Listing 22: Recursive generic `List<T>` definition with constraints

5.5.6. `newtype`

The `newtype` special form introduces a new opaque type whose scope is limited to the given body expression.

```
newtype/3<T, U>(                                                    Signature
  name: GenericIdent,
  value: Type<T>,
  body: Expr<U>
) -> Expr<U>
```

Listing 23: `newtype` special form signature

`newtype` shares a lot of similarity with the `type` special-form; it also allows for generics and their constraints to be defined, and the binder is available inside the `value` argument to allow for self-referential and mutually recursive type definitions.

Unlike `type`, a type defined via `newtype` opts into nominal type checking by defining an opaque type; two opaque types remain distinct even when their underlying structures coincide. Moreover, whereas `let` only introduces a value-universe binder and `type` only introduces a type-universe binder, `newtype` introduces both – under the same name – inside the `body`:

- the opaque type in the *type universe*
- a constructor in the *value universe* that produces values of that type

The constructor's signature depends on the underlying type:

- if the underlying type is `Null`, the constructor takes no arguments;
- otherwise, it takes a single argument of the underlying type

```
["newtype", "List<T: Number>",                                      JSON
    { "#struct": { "value": "T", "next": "Option<List<T>>" } },
["List", {"#struct": {"value": {"#literal": 2}, "next": ["None"]}}]]
```

Listing 24: Recursive generic newtype `List<T>` definition with constraints

5.5.7. use

The `use` special form is used to import items from modules into the accompanying scope of the `body`. Unlike other special forms – whose variants differ by arity – `use` differentiates signatures by the shape of its second argument:

```
use/3<T>(path: Path, items: Ident<*>, body: Expr<T>) -> Expr<T>     Signature
use/3<T>(path: Path, items: List<Ident>, body: Expr<T>) -> Expr<T>
use/3<T>(path: Path, items: Dict<Ident, Ident | _>, body: Expr<T>) -> Expr<T>
```

Listing 25: `use` special form signatures

- **Glob** (`Ident<*>`): If `*` is used, every public item under its original name is imported from the module. It is a shorthand for explicitly specifying all items.
- **Subset** (`List<Ident>`): Only the specified items in the list are imported, with their names unchanged. It is a shorthand for a dictionary where all values are _.
- **Explicit** (`Dict<Ident, Ident | _>`): Each key designates an imported item; the value of each dictionary entry provides the local name. Using _ as a value keeps the name unchanged.

After import, the items are brought into scope by their respective binding and therefore can be referenced as ordinary local names. Although modules are not first-class values, importing the module identifier allows code to avoid referring to it by its absolute path.

```json
["use", "::graph", {"#tuple": ["types"]},
["use", "types::knowledge::entity", {"#tuple": "EntityUuid"}
["EntityUuid", "random_uuid"]]]
```

Listing 26: Nested `use` for relative sub-module access

```json
["use", "core::math", "*",
["add",
  ["sub", "foo", "bar"],
  ["mul", "foo", "baz"]
]]
```

Listing 27: Wildcard (`*`) import of all public items

```json
["use", "core::math", {"#tuple": ["add", "mul", "sub"]},
["add",
  ["sub", "foo", "bar"],
  ["mul", "foo", "baz"]
]]
```

Listing 28: Importing a selected list of items

5.5.8. `fn`

The `fn` special form declares a new closure. A closure is a value that can be invoked with a set of arguments, enabling modular and higher-order code.

```
fn/4<T>(
  generics: Dict<Ident, _ | Type> | List<Ident>,
  parameters: Dict<Ident, Type>,
  returns: Type<T>,
  body: Expr<T>
) -> Callable
-- Callable is an opaque stand-in; the exact function type cannot be
represented in this surface notation.
```

Listing 29: `fn` special form signature

The first argument specifies the generic parameters and has two distinct forms:

- **Implicit** (`List<Ident>`) – a list of identifiers that introduces generic parameters without any bounds. It is a shorthand for a map where each value is _.
- **Explicit** (`Dict<Ident, _ | Type>`) – a dictionary, whose keys are the names of the generic parameters and whose values are their corresponding constraints. If _ is used as a value, no constraint is applied to the generic argument. To allow for recursive and dependent generic definitions, the names of all generic parameters are made available inside the constraint definitions.

`parameters` specifies the parameters of the closure and maps their names to their required type. These types may refer to the previously defined generics.

The `returns` type must match the body's result type. While optional – by setting it to `_` – annotating a closure with an explicit return type allows for additional assurances regarding type safety and change detection for user code.

A closure captures the bindings defined previously within the surrounding scope and has access to these, as well as its parameters, inside the body expression. Because all values are immutable, capturing the surrounding scope does not break referential transparency.

```json
[
  ["fn", {"#tuple": ["T"]}, {"#struct": {"value": "T"}}, "T", "value"],
  {"#literal": 2}
]
```

Listing 30: Generic closure: definition and call

```json
[
  ["fn", {"#tuple": []}, {"#struct": {"value": "_"}}, "_", "value"],
  {"#literal": 2}
]
```

Listing 31: Closure with fully inferred types

5.5.9. `input`

The `input` special form is used for programs to request data outside of normal control flow. It supports the following signatures:

```
input/2<T>(name: Ident, type: Type<T>) -> T
input/3<T>(name: Ident, type: Type<T>, default: T) -> T
```

Listing 32: `input` special form signatures

Conceptually, `input` is akin to dependency injection in Java frameworks such as Spring [62] or to effect handlers in effect-oriented languages [8]: data is supplied by the host environment rather than computed inside the program.

If a `default` value is provided, the input becomes optional; otherwise, it is required whenever the program contains a reachable input call. The compiler itself determines reachability through control-flow analysis from the main entry point. If a specific `input` call cannot be reached (due to, e.g. constant evaluation determining that a branch is always false), it can be safely omitted.

We have chosen to allow side-channel injection of values, as the goal of HashQL is to be primarily a query language. Therefore, the ability to quickly access configuration values at a call-site, instead of needing to thread large parameter structs through every closure, was deemed to be more beneficial than the loss of more explicit data-flow.

Even though `input` allows for interactive values to be specified, referential transparency is still upheld as values are immutable - they cannot be changed once set at the

beginning of execution by the user. Results still depend solely on the immutable input set chosen by the host.

```json
["let", "foo",
  ["input", "bar", "Integer"],
["==", "foo", {"#literal": 42}]]
```

Listing 33: Binding an injected value to a local name

5.5.10. access

The `access` special form is used to retrieve a value from a field inside a struct or an element from a tuple. It has the following signatures:

```
access/2<T: Struct, U>(value: Expr<T>, field: Ident) -> T.field where field ∈ T
access/2<T: Tuple, U>(value: Expr<T>, field: Natural) -> T.field where 0 ≤ field < |T|
```

Listing 34: `access` special form signatures

The selector must be a **literal**, not a computed expression.

- For a struct, `field` must be an identifier token that appears verbatim in source.
- For a tuple, `field` must be an integer literal index and cannot be behind any redirection, such as a variable; tuple indices are zero-indexed.

Any attempt to use a variable, an expression, or any run-time value as a field selector is rejected at compile time during the lowering of the AST. Additionally, during type-checking inside the HIR the compiler will abort compilation and emit a diagnostic if a field or index does not exist.

To make authoring code easier for human users, `access` is also aliased to `.`, meaning that `[".", value, field]` is equivalent to `["access", value, field]`. This is possible because `.` is a valid identifier, as per the grammar specified in Appendix B, and can, like any other special-form or binding, be shadowed by a subsequent local binding.

The semantics of the `access` special form have been influenced by the Rust field access [49:expr.field] and tuple index expressions [49:expr.tuple-index].

```json
[".", "foo", {"#literal": 1}]
```

Listing 35: Accessing the second element in a tuple

```json
[".", "foo", "bar"]
```

Listing 36: Accessing a field in a struct

```
["let", "foo", ["input", "foo", {"#struct": {"bar": "Integer"}}]
[".", "foo", "baz"]
]

// Error: Field 'baz' does not exist
```

Listing 37: Invalid field access (`baz`) triggers a compile-time diagnostic

5.5.11. `index`

The `index` special form is the dynamic counterpart to `access`. It retrieves a value from the dynamically sized collections of the language – notably, dictionaries and lists. It has the following signatures:

```
index/2<K, V>(value: Expr<Dict<K, V>>, index: Expr<K>) -> Option<V>
index/2<V>(value: Expr<List<V>>, index: Expr<Integer>) -> Option<V>
```

Signature

Listing 38: `index` special form signatures

Because the index is calculated at run time and, therefore, dynamic, the compiler cannot guarantee that a lookup succeeds. Thus, the result is always wrapped in `::core::option::Option<V>`.

The design mirrors Rust's fallible `get` methods on collections [47:HashMap::get, 47:slice::get]. We intentionally avoid Rust's array indexing semantics [49:expr.array.index] as these panic on a miss. While panics are admissible and even desired in some system-level code, they are inappropriate in a language designed for querying graph databases. Therefore, HashQL has no concept of panics and can consequently not express Rust's indexing semantics, even if it would be desirable.

To make authoring code easier for human users, `index` is also aliased to `[]`, meaning that `["[]", value, index]` is equivalent to `["index", value, field]`. This is possible because `[]` is a valid identifier, as per the grammar specified in Appendix B, and can, like any other special-form or binding, be shadowed by a subsequent local binding.

Currently, the compiler will not emit any warning if it can statically determine that a dictionary or list index will always be out of range. This might change in a future iteration of the compiler.

```json
["let", "key", {"#literal": "bar"}
["[]", "foo", "key"]]
```

JSON

Listing 39: Indexing a dictionary by key `"bar"`

```json
["let", "index", ["+", {"#literal": 2}, "offset"],
["[]", "foo", "index"]]
```

JSON

Listing 40: Indexing the second element from an offset in a list

5.6. Packages and modules

HashQL organises code within a two-tier namespace hierarchy. A **package** is an isolated compilation unit and in itself a module and therefore addressable. A package may expose additional **modules** that may be nested to an arbitrary depth.

Absolute paths (see Appendix B) start with `::package` (the name of the package), descend through module names, and name an item. For example, `::core::math::sqrt` denotes the item `sqrt` in the module `core::math` within the `core` package.

Circular references between packages or modules are forbidden and will lead to a compilation error. The limitation on cyclic dependencies between modules inside packages may be lifted in the future as outlined in Chapter 8.

5.6.1. Built-in packages

The initial runtime distributes four packages:

- **`kernel`** – special forms and type primitives
- **`core`** – foundational types, their constructors, and intrinsics
- **`graph`** – graph-query specific data structures and intrinsics
- **`main`** – the implicit user entry-point package (exactly one per program)

At present, all user code lives in `main` and contains no sub-modules. Declaring additional packages and modules is not yet supported, but is planned as outlined in Chapter 8.

5.6.1.1. `kernel` package

The `kernel` package is deliberately minimal. It contains exactly two sub-modules:

- **`special_form`** – defines every special form recognized by the compiler (see: Chapter 5.5)
- **`type`** – provides the compiler-defined types, such as `String`, `Integer`, etc (see: Chapter 6)

`kernel` differs from `core`, even though both provide types and intrinsics. Items in `kernel` are **fundamental**: they are required by the compiler itself to function and cannot be re-implemented in user space. This is also reflected in the phase where intrinsics are expanded. Whereas the `kernel` expands intrinsics in the AST, the `core` module expands them later in the HIR as a specialisation pass; until then `core` intrinsics are just ordinary closures.

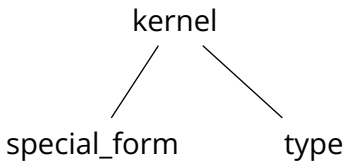


Figure 7: `kernel` Module Layout

5.6.1.2. `core` package

The `core` package contains functionality that is required for HashQL to function, but unrelated to graph querying. Its public surface is divided into the following sub-modules:

- **`bits`** – bit-level integer operations (shift, mask)
- **`bool`** – Boolean combinators: `and`, `or`, `not`
- **`cmp`** – comparison functions such as `>`, `<`, `==`
- **`json`** – JSON utilities and type definitions
- **`math`** – arithmetic on numeric primitives such as `+`, `-`, `*`, `/`
- **`option`** – `Option` type and the `Some`/`None` new-types
- **`result`** – `Result` type and the `Ok`/`Err` new-types
- **`url`** – `Url` new-type and helpers
- **`uuid`** – `Uuid` new-type and helpers

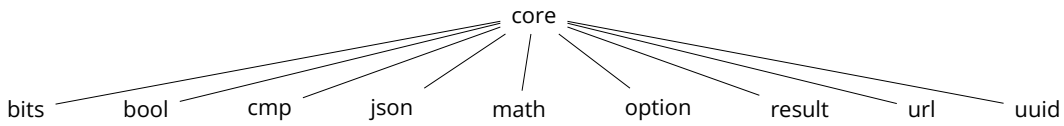


Figure 8: `core` module layout

5.6.1.3. `graph` package

The `graph` package contains everything required to query the bi-temporal graph. Its modules fall into three logical groups:

Query Pipeline

These are functions used to interact with the graph database directly. We structure them into three distinct stages:

- **head** – entry point that **initiates** a query and produces the `Graph<T>` effect type. Head functions determine whether the query starts from entities, entity types, property types, or data types.
- **body** – combinators that **transform** a `Graph<T>` and return a new `Graph<U>` (e.g. `filter`, `flat_map`, `map`, `filter_map`). These functions compose to express any operation on the graph that is to be performed without materialising any intermediate values.
- **tail** – terminal operations that **evaluate** a `Graph<T>` and return a materialised result (e.g. `collect`, `first`, `count`, `exists`)

The decomposition into `head → body → tail` follows the prior work in graph/query systems [61], such as XTQL's source/tail operator distinction and Gremlin's source/step distinction [45].

Utility Modules

For each supported node kind, we expose a set of helper intrinsics to traverse the graph. For example, the `entity` module provides intrinsics for looking up the type of an entity or querying a specific nested property.

In the future, we expect a number of these intrinsics to be replaced by traditional HashQL closures, once the PostgreSQL evaluator is sufficiently extended to support them.

Types

The `types` module mirrors the `@blockprotocol/type-system` Rust crate and contains all the data types required to interface with the graph.

The translation from Rust types to HashQL types is currently maintained manually, with plans to facilitate compile-time reflection tools like facet [5] to automatically generate them, as outlined in Chapter 8.

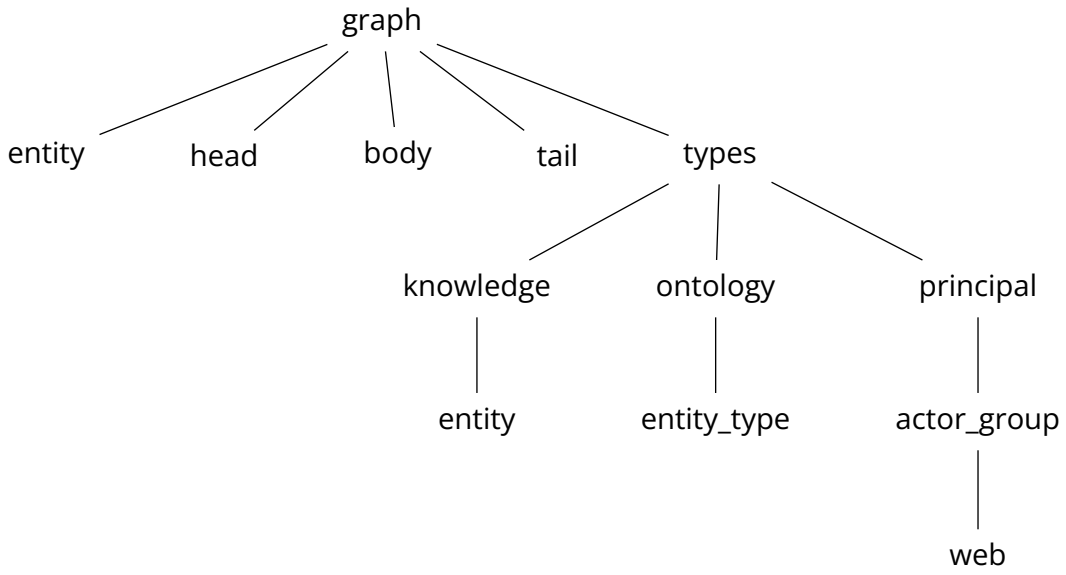


Figure 9: `graph` module layout

5.6.2. Paths and name resolution

Within a HashQL program, an item can be referenced in two ways:

1. **Absolute paths** – begins with `::package`, e.g. `::core::math::add`
2. **Relative paths** – one or more identifiers separated by `::` with no leading `::`, e.g. `add` or `math::add`

A reference to a locally declared variable is simply a single segment relative path.

This design has been primarily influenced by prior work in Rust [49:7.3].

5.6.2.1. Resolution Algorithm

The compiler maintains a single lexical **binding** stack per compilation unit. This is possible since any special form that defines a new entity does so in the supplied body of said closure.

The resolver keeps track of the following meta information during name resolution:

- **name** – the identifier as it appears in the current scope (after any `use` renaming)
- **kind** – constructor, value, module, etc.
- **universe** – value or type universe
- **node** – the data of the item it is referring to

The layout of this information is different and more featureful than the options presented here, but these four properties are used during name resolution.

Package roots occupy the bottom of the stack, so package names automatically participate in relative name lookup as a fallback.

To resolve a relative path P ($S_0 :: S_1 :: \ldots :: S_n$) for a requested universe U using the specified resolver R the compiler performs the algorithm outlined in Algorithm 1.

Algorithm 1: Name Resolution

```
procedure Find-Item(R, U, P)
    ▷ Initialize Variables
    S ← split(P, " :: ")
    C ← R

    ▷ Traverse all segments until the last
    for i < |S| − 1 do
        f ← false
        for j < |C| do
            c ← C_{|C| −j−1}
            if c.name = S_i ∧ c.kind = module then
                C ← c.node.children
                f ← true
                break
            end
        end
        if f = false then
            ▷ Unresolved Module S_i
            terminate
        end
    end

    ▷ Resolve final item
    for j < |C| do
        c ← C_{|C−j−1|}
        if c.name = S_{|S| −1} ∧ c.universe = U then
            return c.node
        end
    end

    ▷ Unresolved Item
    terminate
end
```

```json
["use", "core::oracle", {"#tuple": ["answer"]},
// `answer == 42`
["let", "answer", {"#literal": 24},
// `answer` is being shadowed: `answer`== 24
["==", "answer", "42"]]] // evaluates to false
```

Listing 41: Local `let` shadows an imported binding

```json
["let", "answer", {"#literal": 24},
// `answer == 24`
["use", "core::oracle", {"#tuple": ["answer"]},
// `answer` is being shadowed: `answer`== 42
["==", "answer", "42"]]] // evaluates to true
```

Listing 42: Later `use` shadows an earlier local binding

```json
["==", "core::oracle::answer", "42"] // evaluates to true
```

Listing 43: Direct reference with an absolute package path

5.6.3. Visibility

The four built-in packages (`kernel`, `core`, `graph`, `main`) expose only types, constructors, and intrinsics; they contain no internal implementation that must be hidden. Consequently, every item is currently public.

User-defined packages do not yet exist, and the built-in modules are generated statically by the compiler rather than authored as standalone HashQL programs. They therefore export exactly the items they choose, even when deep cross-module dependencies are present (as illustrated by the `graph` package). For this reason, finer-grained visibility controls have not yet been necessary and therefore have been left unimplemented.

Future revisions will introduce explicit modifiers – `public`, `package`, `module`, `private` – on both packages and individual items. Since today's API is wholly public, adding these modifiers later will be fully backwards-compatible while preserving semantics.

5.6.4. Future `std` package

A future iteration of the compiler will introduce an `std` package, which will re-export the public interface of the built-in modules.

Consolidating the API behind a single, stable package name improves usability while remaining source-compatible: existing programs will continue to compile, and new programs may reference either the individual packages or use the unified entrypoint.

This design follows the precedent set by Rust, where the `std` crate combines the functionality of the `core` and `alloc` crates [47].

6. Type System

The type system introduced in this work is designed to satisfy two overarching goals:

- **Compatibility** – it must be able to express every construct of the HASH type system and model cyclic data structures soundly.
- **Expressiveness** – it must provide the abstractions expected of a modern functional language, including bounded parametric polymorphism, recursive data structures and higher-order functions.

Semantically, the system combines Rust-style algebraic data types [49:10.1] with TypeScript-style structural typing, unions and intersections [35:Object Types]. Its formal underpinnings derive from System F, System $F_{<:}$, and a Hindley-Milner-inspired inference engine with subtyping.

Types are structural by default – required for HASH type system compatibility – but with nominal opt-in through the use of an `Opaque` type wrapper introduced by using the `newtype` special form (Chapter 5.5.6). The structural-nominal trade-off is discussed in Chapter 6.8.

Feature	Description	Section
Lattice	Complete lattice with top `Unknown` ($\top$) and bottom `Never` ($\bot$)	Chapter 6.4
Generics	Bounded parametric polymorphism	Chapter 6.5
Recursive μ-types	Equi-recursive definitions for cyclic data	Chapter 6.6
Variance	Covariant by default; closure parameters contravariant, dictionary keys invariant	Chapter 6.7
Inference	Constraint generation and Hindley-Milner-style unification with subtyping	Chapter 6.10

Table 1: Key capabilities of the type system

6.1. Influences and Prior Work

This work is influenced by – and builds on – classical texts in type theory and the typed λ-calculus, notably Benjamin C. Pierce. [39], Benjamin C. Pierce (Ed.). [40], and Robert Harper. [25]. These works provide the foundational techniques we use and the standard progress/preservation metatheory.

6.2. Notation and Conventions

We introduce the meta-symbols, contexts and judgement forms used in this chapter, as well as their description. Concrete type constructors and specialised operators are introduced in subsequent chapters.

6.2.1. Variables

A variable's letter indicates which universe or role it occupies.

- **Terms / values** – lower-case Latin $e, f, g, \ldots$ (inhabitants of the value universe).
- **Type constants / constructors** – symbols starting with an upper-case Latin letter $\mathrm{Int}, \mathrm{Number}, \ldots$ (inhabitants of the type universe)
- **Bound generics** – lower-case Greek $\alpha, \beta, \gamma, \ldots$ bound generic parameters that appear under a type-level binder $\Lambda\alpha.$ or a universal quantifier $\forall\alpha.$ (formalised in Chapter 6.5).
- **Inference variables** - hatted lower-case Greek $\hat{\alpha}, \hat{\beta}, \hat{\gamma}, \ldots$. These placeholders are created by the inference engine or by the user-written underscore (_) type and are solved during unification.
- **Contexts** – upper-case Greek $\Gamma, \Delta, \Sigma, \ldots$ (see below)

6.2.2. Contexts

A context collects assumptions and is stateful across typing rules.

- **Value context** Γ – maps identifiers to their types
- **Type context** Δ – records every generic parameter and its associated bounds
- **Derivation context** Θ – stores the visited pairs in the current derivation to support recursive types
- **Variance Context** Φ – stores the associated variance for every type

We write $\Gamma, x : T$ to extend a context with a new binding. Unless stated otherwise, we abbreviate the full judgement $\Phi; \Theta; \Delta; \Gamma \vdash e : T$ by only showing the innermost part $\Gamma \vdash e : T$.

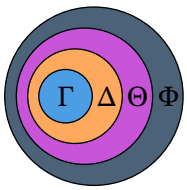


Figure 10: Hierarchy of contexts

We write $\vdash X\ \mathrm{wf}$ to mean that context X is well-formed. Every rule implicitly assumes all mentioned contexts are well-formed.

$$
\begin{aligned}
&\text{Kind} &&:= * \mid \text{Kind} \Rightarrow \text{Kind} \\
&\text{dom}(C) &&:= \{x \mid (x : y) \in C\} \\
&\text{def}(C) &&:= \{(x : y) \mid (x : y) \in C\} && \text{isolate assignment bindings}
\end{aligned}
$$

$$
\begin{aligned}
&\vdash \Delta \text{ wf} \;\Leftrightarrow \\
&\quad \text{finite} \\
&\quad \wedge\, |\text{def}(\Delta)| = |\text{dom}(\text{def}(\Delta))| && \text{unique kind per type variable} \\
&\quad \wedge\, \forall X : K \in \Delta . K \in \text{Kind} \\
&\quad \wedge\, \forall \alpha <: T \in \Delta . \alpha : * \in \Delta \wedge \Delta \vdash T : * \\
&\quad \wedge\, \forall T <: \alpha \in \Delta . \Delta \vdash T : * \wedge \alpha : * \in \Delta \\
&\quad \wedge\, \forall \alpha = T \in \Delta . \alpha : * \in \Delta \wedge \Delta \vdash T : *
\end{aligned}
$$

$$
\begin{aligned}
&\vdash \Gamma \text{ wf} \;\Leftrightarrow \\
&\quad \text{finite} \\
&\quad \wedge\, |\text{def}(\Gamma)| = |\text{dom}(\text{def}(\Gamma))| && \text{unique type per term} \\
&\quad \wedge\, \forall x : T \in \Gamma . \Delta \vdash T : *
\end{aligned}
\tag{4}
$$

$$
\begin{aligned}
&\vdash \Theta \text{ wf} \;\Leftrightarrow \\
&\quad \text{finite} \\
&\quad \wedge\, \forall \langle T_1, T_2 \rangle \in \Theta . \Delta \vdash T_1 : * \wedge \Delta \vdash T_2 : *
\end{aligned}
$$

$$
\begin{aligned}
&\vdash \Phi \text{ wf} \;\Leftrightarrow \\
&\quad \text{finite} \\
&\quad \wedge\, |\text{def}(\Phi)| = |\text{dom}(\text{def}(\Phi))| && \text{unique variance per type variable} \\
&\quad \wedge\, \exists! v_0 . (\varphi : v_0) \in \Phi \wedge v_0 \in \{+1, 0, -1\} \\
&\quad \wedge\, \forall T : v \in (\Phi \setminus (\varphi : v_0)) . v \in \{+1, 0, -1\} \wedge \Delta \vdash T : *
\end{aligned}
$$

Figure 11: Context Invariants

6.2.3. Syntax

$t ::=$		terms:
	x	variable
	$\lambda x : T.t$	abstraction
	$\langle\!\langle l_1 : t_1, ..., l_n : t_n \rangle\!\rangle$	closed struct
	$\langle l_1 : t_1, ..., l_n : t_n \rangle$	open struct
	$(\!(t_0, ..., t_{n-1})\!)$	tuple
	$t\ t$	application
	$t[t]$	subscript
	$t[x := y]$	capture avoiding substitution
	$t.l$	projection
	$t[\![T]\!]$	type application
$\upsilon ::=$		variance
	$+1$	covariant
	-1	contravariant
	0	invariant
$T ::=$		types:
	α	type variable (bound)
	$\hat{\alpha}$	type variable (inference)
	$\Lambda\alpha : K.T$	generic abstraction
	$T\ T$	generic application
	$T[\alpha \mapsto U]$	capture-avoiding substitution
	$\forall\alpha <: T.T$	universal type
	$\mu X.T$	recursive type
	$(\omega \mid l_1 : T_1, ..., l_n : T_n)$	struct
	$T_0 \times ... \times T_{n-1}$	tuple
	$A \to B$	closure
	$o\langle s \mid T \rangle$	nominal
	$T \cup T$	union
	$T \cap T$	intersection
$K ::=$		kinds:
	$*$	fully formed type
	$K \Rightarrow K$	type constructor

$\Gamma ::=$		value context:
	$\varnothing$	empty
	$\Gamma, x : T$	variable binding
$\Delta ::=$		type context:
	$\varnothing$	empty
	$\Delta, \alpha : K$	variable binding
	$\Delta, \alpha <: T$	upper bound
	$\Delta, T <: \alpha$	lower bound
	$\Delta, \alpha = T$	equal bound
$\Phi ::=$		variance context:
	$(\varphi : +1)$	default
	$\Phi, T : \upsilon$	type variance binding
	$\Phi, \varphi : \upsilon$	env variance binding
$\Theta ::=$		derivation context:
	$\varnothing$	empty
	$\Theta, \langle T_1, T_2 \rangle$	insert fresh pair
$R ::=$		relation:
	$T_1 \leq T_2$	declarative subtyping
	$T_1 <: T_2$	variance-aware subtyping
	$T_1 \equiv T_2$	definitional equivalence

Figure 12: Type System Syntax

6.3. Foundation

Definition 1 **Term Lookup** : If a binding $e : T$ appears in the value context Γ, we assume that the term e is assigned to the type T (T-VAR) [39:9.2].

$$\frac{(e : T) \in \Gamma}{\Gamma \vdash \text{e: T}} \quad \text{T-VAR}$$

Definition 2 **Type Lookup** : If a binding $T : K$ appears in the type context Δ, we assume that the type T is assigned to the kind K (K-VAR) [39:29].

$$\frac{(T : K) \in \Delta}{\Delta \vdash \text{T: K}} \quad \text{K-VAR}$$

Definition 3 **Variance Lookup** : The variance context Φ is a finite map $H \mapsto \{+1, -1, 0\}$.

If a binding $T : v$ is present, the lookup returns that variance (V-VAR1). When no binding exists, the lookup yields the default $+1$ (V-VAR2).

The special entry $(\varphi : v)$ inside Φ tracks the current variance during a derivation, and is updated whenever a transition occurs; it defaults to being covariant ($+1$).

$$\frac{(T : v) \in \Phi}{\Phi \vdash \text{T: v}} \quad \text{V-VAR1}$$

$$\frac{(T : v) \notin \Phi}{\Phi \vdash \text{T: } +1} \quad \text{V-VAR2}$$

Definition 4 **Free Variables** : The set of free variables of term t, written $\text{FV}(t)$ is defined inductively [39:5.3.2]:

$$\begin{aligned} \text{FV}(x) &= \{x\} \\ \text{FV}(\lambda x.t_1) &= \text{FV}(t_1) \setminus \{x\} \\ \text{FV}(t_1 t_2) &= \text{FV}(t_1) \cup \text{FV}(t_2) \end{aligned} \tag{5}$$

6.4. Lattice

Let H be the set of every type expressible in HashQL. Equipped with the subtype relation $<:$ ("is a subtype of"), $(H, <:)$ forms a partially ordered set.

We complement this partially ordered set with two binary operations:

- meet ($\sqcap$) – unique infimum (greatest lower bound) between any two elements $\{T_1, T_2\} \in H$
- join ($\sqcup$) – unique supremum (least upper bound) between any two elements $\{T_1, T_2\} \in H$

These operations make H simultaneously a meet- and a join-semilattice; hence $(H, <:, \sqcap, \sqcup)$ forms a lattice.

The lattice is bounded: its top element is `Unknown` ($\top$) and its bottom is `Never` ($\bot$).

Definition 5 **Declarative Subtyping ($\leq$)** : $T_1 \leq T_2$ is the declarative (variance-agnostic) subtyping relation. It is a partial order: reflexive (S-REFL), transitive (S-TRANS), and antisymmetric (S-ANTI) [39:16].

Definition 6 **Subtyping ($<:$)** : $T_1 <: T_2$ is the variance-aware counterpart of $T_1 \leq T_2$. It is parameterised by the variance context Φ : depending on Φ it may check $T_1 \leq T_2$, or $T_2 \leq T_1$ (contravariant position), or require $T_1 \equiv T_2$ (invariant position), as detailed in Chapter 6.7.

When variance is not in effect (no use of Φ along the derivation), $T_1 <: T_2$ coincides with $T_1 \leq T_2$.

Definition 7 **Equivalence ($\equiv$)** : Types are definitionally equivalent if $T_1 <: T_2$ and $T_2 <: T_1$ holds (S-ANTI).

$$T <: T \qquad \text{S-REFL}$$

$$\frac{\mathrm{T} <: \mathrm{U} \quad \mathrm{U} <: \mathrm{V}}{\mathrm{T} <: \mathrm{V}} \qquad \text{S-TRANS}$$

$$\frac{\mathrm{T} <: \mathrm{U} \quad \mathrm{U} <: \mathrm{T}}{\mathrm{T} \equiv \mathrm{U}} \qquad \text{S-ANTI}$$

Definition 8 **Subsumption** : A term of type T can be safely viewed as type U if $T <: U$ holds (T-SUB) [39:15].

$$\frac{\Gamma \vdash \mathrm{e}: \mathrm{T} \quad \mathrm{T} <: \mathrm{U}}{\Gamma \vdash \mathrm{e}: \mathrm{U}} \qquad \text{T-SUB}$$

Definition 9 **Meet ($\sqcap$)** : For any two types $\{T_1, T_2\} \in H$, their meet $T_1 \sqcap T_2$ is the greatest lower bound (GLB): it is a subtype of both (S-MEET1, S-MEET2), and every common subtype of T_1 and T_2 is below it.

The meet operator is monotone (M-MONO), commutative (M-COMM), associative (M-ASSOC), absorptive (M-ABSORB), idempotent (M-IDEM), and distributes over join (M-DIST).

$$T_1 \sqcap T_2 <: T_1 \qquad \text{S-MEET1}$$

$$T_1 \sqcap T_2 <: T_2 \qquad \text{S-MEET2}$$

$$\frac{\mathrm{T} <: U_1 \quad \mathrm{T} <: U_2}{\mathrm{T} <: U_1 \sqcap U_2} \qquad \text{S-MEET3}$$

$$\frac{T_1 <: U_1 \quad T_2 <: U_2}{T_1 \sqcap T_2 <: U_1 \sqcap U_2} \qquad \text{M-MONO}$$

$$T \sqcap U = U \sqcap T \qquad \text{M-COMM}$$

$$T \sqcap (U \sqcap V) = (T \sqcap U) \sqcap V \quad \text{M-ASSOC}$$

$$T \sqcap (T \sqcup U) = T \quad \text{M-ABSORB}$$

$$T \sqcap T = T \quad \text{M-IDEM}$$

$$T \sqcap (U \sqcup V) = (T \sqcap U) \sqcup (T \sqcap V) \quad \text{M-DIST}$$

Definition 10 **Join ($\sqcup$)** : For any two types $\{T_1, T_2\} \in H$, their join $T_1 \sqcup T_2$ is the least upper bound (LUB): it is a supertype of both (S-JOIN1, S-JOIN2, S-JOIN3), and every common supertype of T_1 and T_2 lies above it.

The join operator is monotone (J-MONO), commutative (J-COMM), associative (J-ASSOC), absorptive (J-ABSORB), idempotent (J-IDEM), and distributes over meet (J-DIST).

$$T_1 <: T_1 \sqcup T_2 \quad \text{S-JOIN1}$$

$$T_2 <: T_1 \sqcup T_2 \quad \text{S-JOIN2}$$

$$\frac{U_1 <: \mathrm{T} \qquad U_2 <: \mathrm{T}}{U_1 \sqcup U_2 <: \mathrm{T}} \quad \text{S-JOIN3}$$

$$\frac{T_1 <: U_1 \qquad T_2 <: U_2}{T_1 \sqcup T_2 <: U_1 \sqcup U_2} \quad \text{J-MONO}$$

$$T \sqcup U = U \sqcup T \quad \text{J-COMM}$$

$$T \sqcup (U \sqcup V) = (T \sqcup U) \sqcup V \quad \text{J-ASSOC}$$

$$T \sqcup (T \sqcap U) = T \quad \text{J-ABSORB}$$

$$T \sqcup T = T \quad \text{J-IDEM}$$

$$T \sqcup (U \sqcap V) = (T \sqcup U) \sqcap (T \sqcup V) \quad \text{J-DIST}$$

Definition 11 **Bottom Type ($\bot$)** : $\bot$ (`Never`) is the least element of the lattice: for all types T we assert $\bot <: T$ (S-BOT) [39:15]

$$\bot <: T \quad \text{S-BOT}$$

Definition 12 **Top type ($\top$)** : $\top$ (`Unknown`) is the greatest element of the lattice: for all types T we assert $T <: \top$ (S-TOP) [39:15]

$$T <: \top \quad \text{S-TOP}$$

Lemma 13 **Meet-Join Absorption under Subtyping** :

$$\frac{\mathrm{T} <: \mathrm{U} \qquad \dfrac{}{\mathrm{T} <: \mathrm{T}}\,\text{S-REFL} \qquad \dfrac{}{T_1 \sqcap T_2 <: T_1}\,\text{S-MEET1} \qquad \dfrac{\mathrm{T} <: U_1 \qquad \mathrm{T} <: U_2}{\mathrm{T} <: U_1 \sqcap U_2}\,\text{S-MEET3} \qquad \dfrac{\mathrm{T} <: \mathrm{U} \qquad \mathrm{U} <: \mathrm{T}}{\mathrm{T} \equiv \mathrm{U}}\,\text{S-ANTI}}{\mathrm{T} \sqcap \mathrm{U} \equiv \mathrm{T}}\,\text{M-SUB}$$

$$\frac{\mathrm{T} <: \mathrm{U} \qquad \dfrac{}{\mathrm{T} <: \mathrm{T}}\,\text{S-REFL} \qquad \dfrac{}{T_1 <: T_1 \sqcup T_2}\,\text{S-JOIN1} \qquad \dfrac{U_1 <: \mathrm{T} \qquad U_2 <: \mathrm{T}}{U_1 \sqcup U_2 <: \mathrm{T}}\,\text{S-JOIN3} \qquad \dfrac{\mathrm{T} <: \mathrm{U} \qquad \mathrm{U} <: \mathrm{T}}{\mathrm{T} \equiv \mathrm{U}}\,\text{S-ANTI}}{\mathrm{T} \sqcup \mathrm{U} \equiv \mathrm{U}}\,\text{J-SUB}$$

Proof. Assume $T <: U$.

1. By rule S-MEET1 we have $T \sqcap U <: T$. Because T is itself below both T (S-REFL) and U (assumption), rule S-MEET3 gives $T <: T \sqcap U$. Thus $T \sqcap U$ and T are mutually subtypes, hence equivalent (S-ANTI).
2. Dual: rule S-JOIN2 gives $U <: T \sqcup U$. Because U is itself above both T (assumption) and U (S-REFL), rule S-JOIN3 gives $T \sqcup U <: U$. Thus $T \sqcup U$ and U are both mutually subtypes, hence equivalent (S-ANTI).

□

Definition 14 **Evaluation of Lattice Operations** : Let $\square \in \{\sqcup, \sqcap\}$.

We write $T \square U \Downarrow S$ for the complete big-step evaluation of a lattice operation yielding S.

For the fallback axioms we also use a base evaluator that omits fallback: $T \square U \Downarrow_0 S$ is the same relation with J-FALL and M-FALL removed.

We use $\Downarrow_0$ only in the negative premises of the fallback rules to avoid circularity; $\Downarrow$ is $\Downarrow_0$ extended with those two rules.

Definition 15 **Ascription** : $\mathrm{as}(e, U)$ is the explicit subsumption operation: if e has type T and $T <: U$, then $\mathrm{as}(a : U)$ has type U (T-AS). Operationally, ascription is inert: $\mathrm{as}(e, U)$ evaluates exactly as e.

$$\frac{\Gamma \vdash \mathrm{e} : \mathrm{T} \qquad \mathrm{T} <: \mathrm{U}}{\Gamma \vdash \mathrm{as}(e, U) : \mathrm{U}}\,\text{T-AS}$$

6.5. Generic types

HashQL distinguishes between two classes of type variables. **Bound parameters** ($\alpha, \beta, \gamma, \ldots$) are introduced explicitly by a type-level abstraction $\Lambda\alpha.$ or by universal quantification $\forall\alpha <: U.$; they participate in α-renaming and may carry multiple lower, upper or fixed bounds. A variable with a fixed bound is called skolemised – it must be definitionally equal to the bound supplied. **Inference variables** ($\hat{\alpha}, \hat{\beta}, \hat{\gamma}, \ldots$), by contrast, arise implicitly – either from a user specifying an Underscore (_) type or from the constraint generator itself. Users cannot explicitly define constraints for the hole; instead, each hole is initially assigned the vacuous bound $\bot <: \hat{\omega}_k <: \top$, after which the solver may tighten bounds and even merge two variables – whether explicit or inferred – into the same equivalence class when equality is entailed.

Bound parameters and inference variables occupy the same universe; therefore, the capture-avoiding substitution operator ($T[\alpha \mapsto U]$) may refer to either.

Since HashQL primarily relies on structural equivalence, with nominal opt-in, β-reduction is applied aggressively; most Λ blocks are removed before constraint generation begins. Because of this, we decided against using De-Bruijn indices [13], instead requiring α-renaming, as each reduction step would require repeated re-indexing traversals – particularly costly with our set of features. Instead, the compiler maintains two monotonically increasing counters, one for bound parameters and another for inference holes. Because these numeric identifiers act as unique identifiers, unification reduces to an integer comparison, and algorithms that walk deep, cyclic type graphs can ignore binder stacks entirely.

Definition 16 **Kinds** : A kind classifies type-level expressions. HashQL uses
- the base kind $*$ for fully-formed, term-classifying, types
- arrow kinds $K \Rightarrow K$ for type-level functions (constructors).

A type variable is always assigned kind $*$; it cannot itself be a type-level function. Higher-kinded types, such as `List<T>`, become term-classifying only after they are applied (yielding $List\ T : *$)

Kind arrows associate to the right: $K_1 \Rightarrow K_2 \Rightarrow K_3$ is abbreviated as $K_1 \Rightarrow (K_2 \Rightarrow K_3)$. [39:29]

$$\frac{\Delta, T_1 \colon K_1 \vdash T_2 : K_2}{\Delta \vdash \Lambda\ T_1 \colon K_1.\ T_2 : K_1 \Rightarrow K_2} \quad \text{K-ABS}$$

$$\frac{\Delta \vdash T_1 \colon K_1 \Rightarrow K_2 \qquad \Delta \vdash T_2 \colon K_1}{\Delta \vdash T_1\ T_2 : K_2} \quad \text{K-APP}$$

Definition 17 **Quantification** : New generic parameters are introduced by universal quantification (Q-GEN) or by type-level abstraction (K-ABS). [39:30]

$$\frac{\Delta; \Gamma \vdash \mathrm{e} \colon \tau \qquad \alpha \notin \mathrm{FV}(\tau)}{\Delta; \Gamma \vdash \mathrm{e} \colon \forall\, \alpha <: \top.\ \tau} \quad \text{Q-GEN}$$

Definition 18 **Generic Subtyping** : Universal quantification and type-level constructors do not participate in subtyping, instead the underlying terms are compared, if $T_1 <: T_2$ then $\forall \alpha <: U_1.T_1 <: \forall \alpha <: U_1.T_2$ (Q-SUB, K-SUB). [39:30]

$$\frac{T_1 <: T_2}{(\forall \alpha <: U.T_1) <: (\forall \alpha <: U.T_2)} \quad \text{Q-SUB}$$

$$\frac{\Delta, T_1 : K_1 \vdash T_2 <: T_3}{(\Lambda T_1 : K_1.T_2) <: (\Lambda T_1 : K_1.T_3)} \quad \text{K-SUB}$$

Definition 19 **Inference Holes** : Inference holes are introduced implicitly. They are added whenever the user requests a hole through underscore (_) or when the constraint generator needs a new hole (T-INF).

$$\frac{\hat{\alpha} \notin \mathrm{FV}(\Delta)}{\Delta, \hat{\alpha} : * \vdash \mathrm{wf}} \quad \text{T-INF}$$

Definition 20 **Application** : A polymorphic value can be instantiated with a concrete type as long as the bound is respected (Q-APP). [39:30]

$$\frac{\Delta; \Gamma \vdash \mathrm{e} : \forall\, \alpha <: \mathrm{U}.\ \tau \qquad \Delta \vdash \mathrm{S} <: \mathrm{U}}{\Delta; \Gamma \vdash \mathrm{e}\ [\![S]\!] : \tau[\alpha \mapsto S]} \quad \text{Q-APP}$$

Definition 21 **α-conversion** : Universal quantification and type-level abstractions are insensitive to the choice of binder (Q-CONV, K-CONV).

$$\frac{\Delta, \mathrm{U} : * \vdash \mathrm{T} : \mathrm{K} \qquad \beta \notin \mathrm{FV}(T)}{\Delta \vdash \forall\, \alpha <: \mathrm{U}\ .\ \mathrm{T} \equiv \forall\, \beta <: \mathrm{U}\ .\ \mathrm{T}[\alpha \mapsto \beta]} \quad \text{Q-CONV}$$

$$\frac{\Delta \vdash \mathrm{T} : \mathrm{K} \qquad \beta \notin \mathrm{FV}(T)}{\Delta \vdash \Lambda\, \alpha : \mathrm{K}\ .\ \mathrm{T} \equiv \Lambda\, \beta : \mathrm{K}\ .\ \mathrm{T}[\alpha \mapsto \beta]} \quad \text{K-CONV}$$

Definition 22 **β-reduction** : Application a type-level function β-reduces to its body with the argument substituted (K-REDUC).

$$\frac{\Delta, \mathrm{T} : \mathrm{K} \vdash \mathrm{S} : \mathrm{K}}{\Delta \vdash (\Lambda \alpha : K.T)\ \mathrm{S} \equiv \mathrm{T}[\alpha \mapsto S]} \quad \text{K-REDUC}$$

Definition 23 **⊤/⊥ kinds** : $\top$ and $\bot$ are fully-formed, term-classifying, types (K-TOP, K-BOT).

$$\Delta \vdash \top : * \quad \text{K-TOP}$$

$$\Delta \vdash \bot : * \quad \text{K-BOT}$$

Definition 24 **Bounds** : A type variable may accrue one or more bounds throughout type-inference and checking:

- Lower Bound: $\alpha <: T$ (G-LOWER)
- Upper Bound: $T <: \alpha$ (G-UPPER)
- Equal Bound: $\alpha = T$ (G-EQUAL)

These constraints are additive and are conjoined if multiple constraints are encountered while maintaining a well-formed environment.

$$\frac{\Delta \vdash \mathrm{T} : *}{\Delta, \alpha <: \mathrm{T} \vdash \mathrm{wf}} \quad \text{G-LOWER}$$

$$\frac{\Delta \vdash \mathrm{T} : *}{\Delta, \mathrm{T} <: \alpha \vdash \mathrm{wf}} \quad \text{G-UPPER}$$

$$\frac{\Delta \vdash \mathrm{T} : *}{\Delta, \alpha = \mathrm{T} \vdash \mathrm{wf}} \quad \text{G-EQUAL}$$

Lemma 25 **Kind Substitution** : Substituting a well-kinded type for a type variable preserves the overall kind. [39:30]

$$\frac{\Delta, \alpha\colon K_1 \vdash \mathrm{T}\colon K_2 \qquad \Delta \vdash \mathrm{U}\colon K_1}{\Delta \vdash \mathrm{T}[\alpha \mapsto U] : K_2} \quad \text{K-SUBST}$$

Definition 26 **Join/Meet Kinding** : The lattice operations $\sqcup$ (join) and $\sqcap$ (meet) are defined only for term-classifying types (K-JOIN, K-MEET).

Rationale. The subtyping lattice is defined over term types (see Chapter 6.4), so no partial order is introduced on higher-kinded type constructors; consequently, there is no notion of join/meet above kind $*$.

$$\frac{\Delta \vdash \mathrm{T}\colon * \qquad \Delta \vdash \mathrm{U}\colon *}{\Delta \vdash \mathrm{T} \sqcup \mathrm{U}\colon *} \quad \text{K-JOIN}$$

$$\frac{\Delta \vdash \mathrm{T}\colon * \qquad \Delta \vdash \mathrm{U}\colon *}{\Delta \vdash \mathrm{T} \sqcap \mathrm{U}\colon *} \quad \text{K-MEET}$$

6.6. Recursive types

HashQL adopts **equi-recursive μ-types** and interprets subtyping and equivalence coinductively in the style of Benjamin C. Pierce. [39:21].

An equi-recursive type is definitionally equal to its one-step unfolding (R-UNFOLD). The subtype relation is the greatest fixed point generated by the existing set of rules plus the co-inductive rules S-REC1/S-REC2. `meet` (⊓) and `join` (⊔) defer to these facts (Lemma 13).

Implementation Note. The checker unfolds each operand once per derivation layer; the axioms defined allow for an arbitrary amount of unfolding to preserve soundness and completeness.

Definition 27 **Recursive Type** : $\mu X.T$ binds X in T. A recursive type is contractive if X occurs only under a constructor in T (such as a sum type, or generic); non-contractive forms (e.g. $\mu X.X$) are ill-formed (K-REC).

$$\frac{\Delta, \mathrm{X}\colon * \vdash \mathrm{T}\colon * \qquad (X \text{ occurs contractively in } T)}{\Delta \vdash \mu\ \mathrm{X}.\ \mathrm{T}\colon *} \quad \text{K-REC}$$

Definition 28 **Fold–Unfold Equivalence** : A μ-type and its unfolding are definitionally equal (R-UNFOLD).

$$\frac{}{\Delta \vdash \mu\ \mathrm{X}.\ \mathrm{T} \triangleq \mathrm{T}[X \mapsto \mu X.T]} \quad \text{R-UNFOLD}$$

Definition 29 **Coinductive Subtyping** : Let Θ be the (unordered) set that records every pair $\langle S, T\rangle$ already encountered while checking $\Delta; \Theta \vdash S <: T$.

Two rules realise the co-inductive semantics:

- **F-closure** (S-REC1) When encountering a *fresh* pair $\langle S_1, S_2\rangle$, add it to Θ and compare the unfolded bodies. This satisfies the closure property: the candidate relation is closed under one-step unfolding.
- **F-consistency** (S-REC2) When encountering an existing pair $\langle S_1, S_2\rangle$, immediately accept it, completing bisimulation. No further unfolding is required because all non-recursive premises were verified on the first visit.

Together S-REC1 and S-REC2 implement the standard greatest fixed point: every recursive pair is assumed related (F-closure) and that assumption is justified by a single successful descent through the type constructors (F-consistency).

$$\frac{S_1 \coloneqq \mu\,\mathrm{X}.\,T_1 \qquad S_2 \coloneqq \mu\,\mathrm{X}.\,T_2 \qquad \langle S_1, S_2\rangle \notin \Theta \qquad \Delta, \mathrm{X}\colon *; \Theta, \langle S_1, S_2\rangle \vdash T_1 <: T_2}{\Delta; \Theta \vdash S_1 <: S_2} \text{ S-REC1}$$

$$\frac{\langle S_1, S_2\rangle \in \Theta}{\Delta; \Theta \vdash S_1 <: S_2} \text{ S-REC2}$$

Definition 30 **Join and Meet for Recursive Types** : Lemma 13 can be used for join and meet of two recursive types, as S-REC1 and S-REC2 allow for a subtyping relationship:

- if $S <: T$ then $S \sqcap T = S$ and $S \sqcup T = T$
- if $T <: S$ then $S \sqcap T = T$ and $S \sqcup T = S$
- otherwise: $S \sqcap T = S \wedge T$ and $S \sqcup T = S \vee T$

6.7. Variance

HashQL supports three variances:

- **covariant** ($+$ / $+1$): check $\Delta \vdash S <: T$ (default)
- **contravariant** ($-$ / -1): check $\Delta \vdash T <: S$
- **invariant** (0): check $\Delta \vdash T \equiv S$

The following positions currently deviate from the default:

- closure parameters are contravariant
- dictionary keys are invariant to preserve hashing and equality constraints

A parameter may be annotated with T^0 (T is invariant), T^+ (T is covariant), T^- (T is contravariant) to override the default.

Definition 31 **Variance Lattice** : We define a variance lattice based on strictness with 0 as the supremum and $+1$ as the infimum.

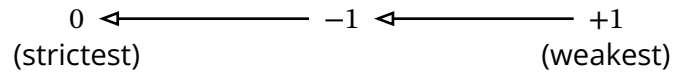


Figure 13: Variance Ordering

We define meet ($v_1 \sqcap v_2$) and join ($v_1 \sqcup v_2$) for the lattice:

$$v_1 \sqcup v_2 = \begin{cases} 0 & \text{if } v_1 = 0 \vee v_2 = 0 \\ -1 & \text{if } v_1 = -1 \vee v_2 = -1 \\ +1 & \text{otherwise} \end{cases}$$
$$v_1 \sqcap v_2 = \begin{cases} +1 & \text{if } v_1 = +1 \vee v_2 = +1 \\ -1 & \text{if } v_1 = -1 \vee v_2 = -1 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

Definition 32 **Type Variance** : Let Φ be the variance context. Define the total lookup $\Phi(T)$ as follows:

$$\Phi(T) = \begin{cases} v & \text{if } (T : v) \in \Phi \\ +1 & \text{otherwise} \end{cases} \tag{7}$$

Because Φ inserts a default entry $(T : +1)$ whenever no explicit binding exists, the function $\Phi(T)$ is total for every well-formed type ($\Delta \vdash T\ \mathrm{wf}$).

Definition 33 **Variance Transition** : We define a function $\mathrm{trans}(c, i)$, which maps the current context variance ($c \in \{-1, \pm 0, +1\}$) and the incoming variance ($i \in \{-1, \pm 0, +1\}$) to the next variance.

As variances are numerically encoded, the function simplifies to:

$$\mathrm{trans}(c, i) = c * i \tag{8}$$

To prove that this is correct, we provide a materialised table of all possible combinations, where the row indicates the current value, whereas the column is the incoming value:

	$+1$	0	-1
$+1$	$+1$	0	-1
0	0	0	0
-1	-1	0	$+1$

Table 2: Variance Transition Matrix

Definition 34 **Variance Flow** : We define a companion function $\mathrm{flow}(c, i)$, which instructs the subtyping algorithm whether to keep the order ($+1$), swap the order (-1), or transition to equivalence checks (± 0).

The function combines the current context variance ($c \in \{-1, \pm 0, +1\}$) with the incoming variance ($i \in \{-1, \pm 0, +1\}$). Whereas Equation 8 returns the effective variance, flow encodes the state change required to achieve that variance. The context keeps track of flips relative to the original derivation (which is covariant). Thus, a contravariant context (-1) paired with a contravariant slot (-1) must return -1 as we transition from contravariance to covariance. ($\mathrm{trans}(-1, -1) = +1$).

To encode this behaviour, we define flow as follows:

$$\mathrm{flow}(c, i) = \begin{cases} 0 & \text{if } \mathrm{trans}(c, i) = 0 \\ -1 & \text{if } \mathrm{trans}(c, i) \neq c \\ +1 & \text{if } \mathrm{trans}(c, i) = c \end{cases} \tag{9}$$

	-1	0	$+1$
-1	-1	0	$+1$
0	0	0	0
$+1$	-1	0	$+1$

Table 3: Variance Flow Matrix

Table 3 shows a materialized view of $\mathrm{trans}(c, i)$ with all possible values. The row indicates the current value and the column the incoming value.

Definition 35 **Variance Subtyping** : We define $<:$ as the variance-aware subtyping judgement and $\leq$ for the underlying variance-unaware subtyping judgement.

$$\begin{aligned} \mathrm{trans}_\Phi(T, U) &= \mathrm{trans}(\Phi(\varphi), \Phi(T) \sqcap \Phi(U)) \\ \mathrm{flow}_\Phi(T, U) &= \mathrm{flow}(\Phi(\varphi), \Phi(T) \sqcap \Phi(U)) \end{aligned}$$

$$T <: U = \begin{cases} \Phi, \varphi : \mathrm{trans}_\Phi(T, U); \Gamma \vdash T \leq U & \text{if } \mathrm{flow}_\Phi(T, U) = +1 \\ \Phi, \varphi : \mathrm{trans}_\Phi(T, U); \Gamma \vdash U \leq T & \text{if } \mathrm{flow}_\Phi(T, U) = -1 \\ \Phi, \varphi : \mathrm{trans}_\Phi(T, U); \Gamma \vdash T \equiv U & \text{otherwise} \end{cases} \tag{10}$$

By recording each type's variance directly in Φ – rather than on constructors – HashQL can treat nominal and structural types uniformly. For nominal and closure forms, the checker first updates the context-variance entry φ in Φ to the declared parameter variance, then descends; structural types skip that update. In either case, subtyping proceeds by applying Equation 10 pointwise to each component.

6.8. Structural vs. Nominal

HashQL utilises structural equivalence by default; a type can obtain a nominal identity through the use of an opaque type. We write $o\langle s | T \rangle$ for an opaque type wrapping T and carrying a unique identifier s.

Definition 36 **Nominal Subtyping** : Two opaques are related only when both their identifiers match and the underlying shapes are in a subtyping relationship (S-NOM).

$$\frac{s_1 = s_2 \qquad \mathrm{T} <: \mathrm{U}}{o\langle s_1|T\rangle <: o\langle s_2|U\rangle} \text{S-NOM}$$

The interplay of structural and nominal types allows us to encode sum-types without extra types or syntax: declare each variant as an opaque and form their structural union (Listing 44, Listing 45). Because the outer layer remains structural, control-flow special-forms such as `if` (Chapter 5.5.2) can narrow the union by eliminating impossible variants – behaviour unavailable in Rust-style nominal enumerations.

```json
["newtype", "Ok<T>", "T",
["newtype", "Err<E>", "E",
["type", "Result<T, E>", ["|", "Ok<T>", "Err<E>"],
{"#literal": null}]]]
```

Listing 44: `Result` encoded with two opaques

$$\begin{aligned}
\Delta \vdash \mathrm{Err} &: * \Rightarrow * \\
\Delta \vdash \mathrm{Ok} &: * \Rightarrow * \\
\Delta \vdash \mathrm{Result} &: * \Rightarrow * \Rightarrow * \\
\mathrm{Ok} &= \Lambda T.o\langle \texttt{::core::result::Ok}|T\rangle \\
\mathrm{Err} &= \Lambda E.o\langle \texttt{::core::result::Err}|E\rangle \\
\mathrm{Result} &= \Lambda T.\Lambda E.(\mathrm{Ok}\ T) \cup (\mathrm{Err}\ E)
\end{aligned} \tag{11}$$

Listing 45: `Result` encoded as a type

Definition 37 **Nominal Join/Meet** : If two opaques share the same identifier, join and meet are delegated to their enclosed types. For monomorphic opaques, this is a no-op; for polymorphic ones, it avoids proliferation of outer-level union/intersection wrappers (J-NOM, M-NOM).

$$\frac{s_1 = s_2}{o\langle s_1|T\rangle \sqcup o\langle s_2|U\rangle = o\langle s_1|T \sqcup U\rangle} \text{J-NOM}$$

$$\frac{s_1 = s_2}{o\langle s_1|T\rangle \sqcap o\langle s_2|U\rangle = o\langle s_1|T \sqcap U\rangle} \text{M-NOM}$$

Definition 38 **Nominal $\bot$-annihilation** : If the inner type is $\bot$, the whole type is $\bot$ (E-NOM).

$$\frac{\mathrm{N} := o\langle s|\bot\rangle}{\mathrm{N} = \bot} \text{E-NOM}$$

6.9. Types and their Rules

We classify types into the following families:

- **Primitives** (Chapter 5.4.3.1): atomic, built-in types.

- **Structs** (Chapter 5.4.3.2): heterogeneous, fixed-field maps from identifiers to types.
- **Tuples** (Chapter 5.4.3.3): heterogeneous, fixed-length ordered sequences.
- **Intrinsics** (Chapter 5.4.3.4, Chapter 5.4.3.5): homogeneous, dynamically sized containers.
- **Closures** (Chapter 5.5.8): arrow types $S \to T$ (first-class function types).
- **Union**: explicit join $S \sqcup T$ – a term inhabits it iff it inhabits S or T; at the type level, it is equivalent to an unmaterialised lattice join (Chapter 6.4).
- **Intersection**: explicit meet $S \sqcap T$ – a term inhabits it iff it inhabits both S and T; at the type level, it is equivalent to an unmaterialised lattice meet (Chapter 6.4).

6.9.1. Primitives

The language defines five primitives in the type system analogous to Chapter 5.4.3.1:

- **Integer**: arbitrary-precision integer
- **Number**: arbitrary-precision floating-point value
- **Boolean**: true or false
- **Null**: explicit absence of a value
- **String**: arbitrary-length ordered sequence of Unicode scalar values

Definition 39 **Primitive Kinds** : All primitive types are fully formed (term-classifying) types (K-STR, K-INT, K-NUM, K-BOOL, K-NULL).

$$\frac{}{\Delta \vdash \mathit{String} : *}\ \text{K-STR} \qquad \frac{}{\Delta \vdash \mathit{Integer} : *}\ \text{K-INT} \qquad \frac{}{\Delta \vdash \mathit{Number} : *}\ \text{K-NUM}$$

$$\frac{}{\Delta \vdash \mathit{Boolean} : *}\ \text{K-BOOL} \qquad \frac{}{\Delta \vdash \mathit{Null} : *}\ \text{K-NULL}$$

Definition 40 **Primitive Subtyping** : Distinct primitives are unrelated except that `Integer` is a subtype of `Number` (S-INT). No other primitive subtyping holds.

$$\frac{\dfrac{}{\Delta \vdash \mathit{Integer} : *}\ \text{K-INT} \qquad \dfrac{}{\Delta \vdash \mathit{Number} : *}\ \text{K-NUM}}{\mathit{Integer} <: \mathit{Number}}\ \text{S-INT}$$

Definition 41 **Boolean Values** : `Boolean` has exactly two inhabitants: `true` and `false` (T-TRUE, T-FALSE).

$$\frac{\dfrac{}{\Delta \vdash \mathit{Boolean} : *}\ \text{K-BOOL}}{\Gamma \vdash \text{true} : \mathit{Boolean}}\ \text{T-TRUE}$$

$$\frac{\dfrac{}{\Delta \vdash \mathit{Boolean} : *}\,\text{K-BOOL}}{\Gamma \vdash \text{false}: \mathit{Boolean}} \qquad \text{T-FALSE}$$

Definition 42 **Null Value** : `Null` has a single inhabitant, `null` (T-NULL).

$$\frac{\dfrac{}{\Delta \vdash \mathit{Null} : *}\,\text{K-NULL}}{\Gamma \vdash \text{null}: \mathit{Null}} \qquad \text{T-NULL}$$

Definition 43 **Literal Typing** : Numeric and string literals inhabit their corresponding primitives (T-INTLIT, T-NUMLIT, T-STRLIT).

$$\frac{\dfrac{}{\Delta \vdash \mathit{Integer} : *}\,\text{K-INT} \qquad \text{n} \in \mathbb{Z}}{\Gamma \vdash \text{n}: \mathit{Integer}} \qquad \text{T-INTLIT}$$

$$\frac{\dfrac{}{\Delta \vdash \mathit{Number} : *}\,\text{K-NUM} \qquad \text{x} \in \mathbb{Q} \setminus \mathbb{Z}}{\Gamma \vdash \text{x}: \mathit{Number}} \qquad \text{T-NUMLIT}$$

$$\frac{\dfrac{}{\Delta \vdash \mathit{String} : *}\,\text{K-STR} \qquad \text{s is a string literal}}{\Gamma \vdash \text{s}: \mathit{String}} \qquad \text{T-STRLIT}$$

6.9.2. Structs

A struct type is a finite set of label–type pairs with pairwise distinct labels, annotated by a width variance ω: $(\omega \mid l_1 : T_1, \ldots, l_n : T_n)$. We refer to covariant-width structs as being "open", whereas invariant-width structs are "closed".

Definition 44 **Struct Width Variance** : The width ranges over the $\{+1, 0\}$ sublattice of the variance lattice (Definition 31):

- $+1$: **open** (width-covariant) – larger domains may be subtypes of smaller ones
- 0: **closed** (width-invariant) – domains must match

Formally, $\omega \in \{0, +1\}$

Definition 45 **Struct Operators** : Let $S \coloneqq (\omega \mid \ell_1 : T_1, \ldots, \ell_n : T_n)$. We define:

- $\mathrm{dom}(S)$ – the set of field labels.
- $\mathrm{rng}(S)$ – the set of field types.
- $|S|$ – the number of fields

Formally:

$$\begin{aligned} \mathrm{dom}(S) &= \{\ell \mid \exists T.\langle \ell, T\rangle \in S\} \\ \mathrm{rng}(S) &= \{T \mid \exists \ell.\langle \ell, T\rangle \in S\} \\ |S| &= |\mathrm{dom}(S)| \end{aligned} \tag{12}$$

Assumption 46 **Distinct Field Labels** : Struct entries are functional by label: each label determines a unique type.

$$\forall \ell \in \mathrm{dom}(S).\exists! T.(\ell, T) \in S \tag{13}$$

Definition 47 **Type Projection** : We introduce type projection as a partial function on types and keys (labels or indices):

$$\begin{aligned} \mathrm{Key} &\coloneqq \mathrm{Ident} \uplus \mathbb{Z} \\ \mathrm{proj} &: H \times \mathrm{Key} \rightharpoonup H \end{aligned} \tag{14}$$

Definition 48 **Struct Projection** : We extend the partial function proj (Definition 47) – assuming Assumption 46 – to include field projection on structs, exactly when the field is present:

$$\forall S = (\omega \mid i_1 : T_1, ..., i_n : T_n).\forall \ell \in \mathrm{dom}(S).\ \mathrm{proj}(S, \ell) = \iota U.\langle \ell, U \rangle \in S \tag{15}$$

(Here, ι is the definite description operator: $\iota x.P(x)$ denotes "the unique x such that $P(x)$" and is defined only when $\exists! x.P(x)$ holds.)

Definition 49 **Struct Kind** : A struct is a fully-formed (term-classifying) type (K-STRC).

$$\frac{}{\Delta \vdash (\omega \mid l_1 : T_1, ..., l_n : T_n) : *} \text{K-STRC}$$

Definition 50 **Struct Subtyping** : The supertype's width determines struct subtyping:

- Closed supertype (0): domains must match and fields compare point-wise (S-STRC1).
- Open supertype (+1): width-covariant; a subtype may have a superset of fields, and shared fields compare point-wise (S-STRC2).

$$\frac{S_1 \coloneqq (0 \mid \ell_1 : T_1, ..., \ell_n : T_n) \quad S_2 \coloneqq (0 \mid m_1 : U_1, ..., m_n : U_n) \quad \mathrm{dom}(S_1) = \mathrm{dom}(S_2) \quad \forall\, \ell \in \mathrm{dom}(S_1).\ \mathrm{proj}(S_1, \ell) <: \mathrm{proj}(S_2, \ell)}{S_1 <: S_2} \text{S-STRC1}$$

$$\frac{S_1 \coloneqq (\omega_1 \mid \ell_1 : T_1, ..., \ell_n : T_n) \quad S_2 \coloneqq (+1 \mid m_1 : U_1, ..., m_n : U_n) \quad \mathrm{dom}(S_1) \supseteq \mathrm{dom}(S_2) \quad \forall\, \ell \in \mathrm{dom}(S_2).\ \mathrm{proj}(S_1, \ell) <: \mathrm{proj}(S_2, \ell)}{S_1 <: S_2} \text{S-STRC2}$$

Definition 51 **Struct Join** : When the domains match, the lattice join is defined fieldwise, and the result's width is the variance-lattice meet of the operands' widths: $\omega_{\text{res}} = \omega_1 \sqcap \omega_2$ (J-STRC).

Restricting the variance preorder (Definition 31) to width values gives $+1\,(\text{open}) \leq 0\,(\text{closed})$, so $+1 \sqcap 0 = +1$ (open) while $+1 \sqcup 0 = 0$ (closed). A closed result would fail to be an upper bound of an open operand in HashQL, since $(+1 \mid \ldots) <: (0 \mid \ldots)$ is not derivable. Choosing width-meet yields an open result and preserves the upper-bound property required for join to be the least upper bound.

If the domains differ, the join is left as an explicit union (no structural alignment). This avoids surprising "partial" joins that would only combine common fields.

$$\frac{S_1 \coloneqq (\omega_1 \mid \ell_1 : T_1, \ldots, \ell_n : T_n) \qquad S_2 \coloneqq (\omega_2 \mid m_1 : U_1, \ldots, m_n : U_n) \qquad \text{dom}(S_1) = \text{dom}(S_2)}{S_1 \sqcup S_2 = (\omega_1 \sqcap \omega_2 \mid \ell_1 : T_1 \sqcup U_1, \ldots, \ell_n : T_n \sqcup U_n)} \text{ J-STRC}$$

Definition 52 **Struct Meet** : The meet of structs is defined casewise by width and domain:

- open ⊓ open (M-STRC1): the result is open; shared fields meet pointwise; fields exclusive to either side are copied unchanged into the result.
- closed ⊓ closed (M-STRC2): the result is closed and defined only when domains match; fields meet pointwise.
- closed ⊓ open (M-STRC3): Defined iff $\text{dom}(\text{open}) \subseteq \text{dom}(\text{closed})$; the result is closed with domain $\text{dom}(\text{closed})$. Shared fields meet pointwise; fields exclusive to the closed side are copied unchanged. The subset precondition is required for the result to be a greatest lower bound: to subtype a closed operand, the meet must be closed with the same domain; to subtype the open operand, having a superset domain suffices.

$$\frac{\begin{array}{c} S_1 \coloneqq (+1 \mid \ell_1 : T_1, \ldots, \ell_n : T_n) \qquad S_2 \coloneqq (+1 \mid m_1 : U_1, \ldots, m_n : U_n) \\ \text{F} \coloneqq \{\langle \ell, \text{proj}(S_1, \ell) \sqcap \text{proj}(S_2, \ell)\rangle \mid \ell \in \text{dom}(S_1) \cap \text{dom}(S_2)\} \\ \text{G} \coloneqq \{\langle \ell, \text{proj}(S_1, \ell)\rangle \mid \ell \in \text{dom}(S_1) \setminus \text{dom}(S_2)\} \\ \text{H} \coloneqq \{\langle \ell, \text{proj}(S_2, \ell)\rangle \mid \ell \in \text{dom}(S_2) \setminus \text{dom}(S_1)\} \end{array}}{S_1 \sqcap S_2 = (+1 \mid F \cup G \cup H)} \text{ M-STRC1}$$

$$\frac{S_1 \coloneqq (0 \mid \ell_1 : T_1, \ldots, \ell_n : T_n) \qquad S_2 \coloneqq (0 \mid m_1 : U_1, \ldots, m_n : U_n) \qquad \text{dom}(S_1) = \text{dom}(S_2)}{S_1 \sqcap S_2 = (0 \mid \langle \ell, \text{proj}(S_1, \ell) \sqcap \text{proj}(S_2, \ell)\rangle \mid \ell \in \text{dom}(S_1) \cap \text{dom}(S_2))} \text{ M-STRC2}$$

$$\frac{\begin{array}{c} S_1 \coloneqq (0 \mid \ell_1 : T_1, \ldots, \ell_n : T_n) \qquad S_2 \coloneqq (+1 \mid m_1 : U_1, \ldots, m_n : U_n) \qquad \text{dom}(S_2) \subseteq \text{dom}(S_1) \\ \text{F} \coloneqq \{\langle l, \text{proj}(S_1, \ell) \sqcap \text{proj}(S_2, \ell)\rangle \mid \ell \in \text{dom}(S_1) \cap \text{dom}(S_2)\} \\ \text{G} \coloneqq \{\langle l, \text{proj}(S_1, \ell)\rangle \mid \ell \in \text{dom}(S_1) \setminus \text{dom}(S_2)\} \end{array}}{S_1 \sqcap S_2 = (0 \mid F \cup G)} \text{ M-STRC3}$$

Definition 53 **Struct ⊥-annihilation** : If any field type is ⊥, the whole struct is ⊥ (E-STRC).

$$\frac{\text{S} \coloneqq (\omega \mid \ell_1 : T_1, \ldots, \ell_n : T_n) \qquad \bot \in \text{rng}(S)}{\text{S} \equiv \bot} \text{ E-STRC}$$

Definition 54 **Struct Literals** : A struct literal typechecks when each field has the stated type and the labels are pairwise distinct; the bracket form determines the width-variance:

- closed (T-STRC1) uses $\langle\!\langle \ldots \rangle\!\rangle$
- open (T-STRC2) uses $\langle \ldots \rangle$

The case $n = 0$ (empty struct) is permitted.

$$\frac{\Gamma \vdash e_1 \colon T_1 \quad \ldots \quad \Gamma \vdash e_n \colon T_n \quad |\{\ell_1, \ldots, \ell_n\}| = \mathrm{n}}{\Gamma \vdash \langle\!\langle l_1 : e_1, \ldots, l_n : e_n \rangle\!\rangle \colon (0 \mid \ell_1 : T_1, \ldots, \ell_n : T_n)} \quad \text{T-STRC1}$$

$$\frac{\Gamma \vdash e_1 \colon T_1 \quad \ldots \quad \Gamma \vdash e_n \colon T_n \quad |\{l_1, \ldots, l_n\}| = \mathrm{n}}{\Gamma \vdash \langle l_1 : e_1, \ldots, l_n : e_n \rangle \colon (+1 \mid \ell_1 : T_1, \ldots, \ell_n : T_n)} \quad \text{T-STRC2}$$

Definition 55 **Term Projection** : Projection at the term level is admissible whenever type projection is defined: if $\Gamma \vdash e : T$ and $k \in \mathrm{dom}(T)$, then $\Gamma \vdash e.k : \mathrm{proj}(T, k)$ (T-PROJ).

$$\frac{\Gamma \vdash \mathrm{e} \colon \mathrm{T} \quad (T, \ell) \in \mathrm{dom(proj)}}{\Gamma \vdash \mathrm{e}.\ell \colon \mathrm{proj}(S, \ell)} \quad \text{T-PROJ}$$

Definition 56 **Field Projection** : Given $S \coloneqq (\omega \mid l_1 : T_1, \ldots, l_n : T_n)$ and $\Gamma \vdash e : S$, field selection is well-typed whenever the label exists: if $f \in \mathrm{dom}(S)$ then $\Gamma \vdash e.f : \mathrm{proj}(S, f)$ (T-PROJ).

6.9.3. Tuples

A tuple type is a finite, ordered sequence of types, which is invariant concerning its width.

Definition 57 **Tuple Operators** : Let $S \coloneqq T_0 \times \ldots \times T_{n-1}$. We define:

- $\mathrm{dom}(S)$ – the valid indices.
- $\mathrm{rng}(S)$ – the set of element types.
- $\mathrm{graph}(S)$ – the index-type graph
- $|S|$ – the number of element

Formally:

$$\begin{aligned} |S| &= n \\ \mathrm{dom}(S) &= \{0, \ldots, n-1\} \\ \mathrm{rng}(S) &= \{T_i \mid i \in \mathrm{dom}(S)\} \\ \mathrm{graph}(S) &= \{\langle i, T_i \rangle \mid i \in \mathrm{dom}(S)\} \end{aligned} \tag{16}$$

Definition 58 **Tuple Projection** : We extend the partial function proj (Definition 47) to include element projection on tuples, exactly when the index is in bounds:

$$\forall S = T_0 \times ... \times T_{n-1}.\forall i \in \mathrm{dom}(S).\ \mathrm{proj}(S, i) = \iota U.(i, U) \in \mathrm{graph}(S) \tag{17}$$

(Here, ι is the definite description operator: $\iota x.P(x)$ denotes "the unique x such that $P(x)$" and is defined only when $\exists! x.P(x)$ holds.)

Definition 59 **Tuple Kind** : A tuple is a fully-formed (term-classifying) type (K-TUP).

$$\frac{}{\Delta \vdash T_1 \times ... \times T_n : *} \quad \text{K-TUP}$$

Definition 60 **Tuple Subtyping** : Tuples subtype only when their domains match (equal arity). Elements are then compared pointwise (S-TUP).

$$\frac{S_1 := T_0 \times ... \times T_{n-1} \quad S_2 := U_0 \times ... \times U_{n-1} \quad \mathrm{dom}(S_1) = \mathrm{dom}(S_2) \quad \forall\, \mathrm{i} \in \mathrm{dom}(S_1) : T_i <: U_i}{S_1 <: S_2} \quad \text{S-TUP}$$

Definition 61 **Tuple Join** : Tuples have fixed arity; join is defined only when the index domains coincide. Their elements are joined pointwise. Otherwise (arity mismatch), the result is $\bot$ by the general disjoint-intersection rule (no structural alignment) (M-TUP).

$$\frac{S_1 := T_0 \times ... \times T_{n-1} \quad S_2 := U_0 \times ... \times U_{n-1} \quad \mathrm{dom}(S_1) = \mathrm{dom}(S_2)}{S_1 \sqcup S_2 = (T_0 \sqcup U_0 \times ... \times T_{n-1} \sqcup U_{n-1})} \quad \text{J-TUP}$$

Definition 62 **Tuple Meet** : Tuples have fixed arity; meet is defined only when index domains coincide. Their elements are met pointwise. With an arity mismatch, the resulting type degenerates to a $\bot$ (no structural alignment) (M-TUP).

$$\frac{S_1 := T_0 \times ... \times T_{n-1} \quad S_2 := U_0 \times ... \times U_{n-1} \quad \mathrm{dom}(S_1) = \mathrm{dom}(S_2)}{S_1 \sqcap S_2 = (T_0 \sqcap U_0 \times ... \times T_{n-1} \sqcap U_{n-1})} \quad \text{M-TUP}$$

Definition 63 **Tuple $\bot$-annihilation** : If any element type is $\bot$, the whole tuple is $\bot$ (E-TUP).

$$\frac{\mathrm{S} := T_0 \times ... \times T_{n-1} \quad \bot \in \mathrm{rng}(S)}{\mathrm{S} \equiv \bot} \quad \text{E-TUP}$$

Definition 64 **Tuple Literals** : A tuple literal typechecks when each element has the stated type.

The case $n = 0$ (empty tuple) is permitted.

$$\frac{\Gamma \vdash e_0 : T_0 \quad ... \quad \Gamma \vdash e_{n-1} : T_{n-1}}{\Gamma \vdash (\!|e_0, ..., e_{n-1}|\!) : T_0 \times ... \times T_{n-1}} \quad \text{T-TUP}$$

Definition 65 **Element Projection** : Given $S \coloneqq T_0 \times ... \times T_{n-1}$ and $\Gamma \vdash e : S$, element selection is well-typed whenever the index exists: if $i \in \mathrm{dom}(S)$ then $\Gamma \vdash e.i : \mathrm{proj}(S, i)$ (T-PROJ).

6.9.4. Closures

A closure type $A \to B$ classifies callables that accept an A and produce a B. We treat the arrow as right-associative and encode multiple parameters via iterated arrows: $T_0 \to ... \to T_{\{n-1\}} \to U$ abbreviates $T_0 \to (... \to (T_{\{n-1\}} \to U))$. Thus, a surface type like `max(Integer, Integer) -> Integer` is written $Integer \to Integer \to Integer$. This keeps the core calculus unary, makes subtyping laws compositional and allows currying later without changing the metatheory.

Definition 66 **Closure Kind** : Closure types are fully formed (term-classifying) types and require term-classifying components (K-CLO).

$$\frac{\Delta \vdash \mathrm{A} : * \qquad \Delta \vdash \mathrm{B} : *}{\Delta \vdash \mathrm{A} \to \mathrm{B} : *} \quad \text{K-CLO}$$

Definition 67 **Closure Subtyping** : Closures are contravariant in their domain and covariant in their codomain (S-CLO).

$$\frac{S_1 \coloneqq A_1 \to B_1 \qquad S_2 \coloneqq A_2 \to B_2 \qquad A_2 <: A_1 \qquad B_1 <: B_2}{S_1 <: S_2} \quad \text{S-CLO}$$

Definition 68 **Closure Arity** : The arity of a closure is the number of right-nested arrows. Define inductively:

$$\begin{aligned} \mathrm{ari}(A \to B) &= 1 + \mathrm{ari}(B) \\ \mathrm{ari}(T) &= 0 \text{ if } \nexists A, B. T = A \to B \end{aligned} \tag{18}$$

Lemma 69 **Closure Arity Propagation** :

$$\mathrm{ari}(A_1 \to B_1) = \mathrm{ari}(A_2 \to B_2) \Rightarrow \mathrm{ari}(B_1) = \mathrm{ari}(B_2) \tag{19}$$

Definition 70 **Closure Join** : We only define join for closures of equal arity. The domain is met (contravariance), and the codomain is joined. When arities differ, the join remains an explicit union (no structural alignment) (J-CLO).

In a future iteration, the equal-arity requirement might be lifted to enable currying and yield tighter least upper bounds (LUBs) (see Chapter 8).

$$\frac{S_1 \coloneqq A_1 \to B_1 \qquad S_2 \coloneqq A_2 \to B_2 \qquad \mathrm{ari}(S_1) = \mathrm{ari}(S_2)}{S_1 \sqcup S_2 = (A_1 \sqcap A_2) \to (B_1 \sqcup B_2)} \quad \text{J-CLO}$$

Definition 71 **Closure Meet** : We only define meet for closures of equal arity. The domain is joined (contravariance), and the codomain is met. When arities differ, the meet degenerates to a $\bot$ (no structural alignment) (M-CLO).

In a future iteration, the equal-arity requirement might be lifted to enable currying and yield tighter greatest lower bounds (GLBs) (see Chapter 8).

$$\frac{S_1 \coloneqq A_1 \to B_1 \qquad S_2 \coloneqq A_2 \to B_2 \qquad \mathrm{ari}(S_1) = \mathrm{ari}(S_2)}{S_1 \sqcap S_2 = (A_1 \sqcup A_2) \to (B_1 \sqcap B_2)} \tag{M-CLO}$$

Definition 72 **Closure Abstraction** : A lambda is well-typed when its body is well-typed under an assumed parameter (T-ABS) [39:9].

$$\frac{\Gamma, \mathrm{x}\colon \mathrm{A} \vdash \mathrm{e}\colon \mathrm{B}}{\Gamma \vdash \lambda\, \mathrm{x}.\ \mathrm{e}\colon \mathrm{A} \to \mathrm{B}} \tag{T-ABS}$$

Definition 73 **Closure Application** : Application checks the argument against the domain via subtyping and yields the codomain (T-APP). For multiple arguments, apply this rule repeatedly [39:9].

$$\frac{\Gamma \vdash \mathrm{f}\colon \mathrm{A} \to \mathrm{B} \qquad \Gamma \vdash \mathrm{a}\colon A' \qquad A' <\colon \mathrm{A}}{\Gamma \vdash (f\,a)\colon \mathrm{B}} \tag{T-APP}$$

6.9.5. Intrinsics

We treat the dynamically sized, homogeneous collections $List\ T$ and $Dict\ K\ V$ as intrinsic type constructors rather than ordinary opaque types. The motivation is the subscript operator $t[k]$: the core calculus currently lacks a general mechanism (e.g., type-class–based resolution) for subscripting. Therefore, indexing is special-cased and handled directly by the type system for these constructors only; subscripting is undefined for all other types. All other containers (e.g., $Option\ T$, $Graph\ T$) remain ordinary types. This special treatment is provisional; see Chapter 8.

Definition 74 **List Kind** : $List$ is a unary type constructor. When applied to a term-classifying type T, the result $List\ T$ is term-classifying (K-LIST).

$$\frac{}{\Delta \vdash \mathrm{List}\colon * \Rightarrow *} \tag{K-LIST}$$

Definition 75 **Dict Kind** : $Dict$ is a binary type constructor. When applied to term-classifying key and value types K and V, the result $Dict\ K\ V$ is term-classifying (K-DICT).

$$\frac{}{\Delta \vdash \mathrm{Dict}\colon * \Rightarrow * \Rightarrow *} \tag{K-DICT}$$

Definition 76 **List Subtyping** : Lists are covariant in their element type (S-LIST).

$$\frac{T_1 <: T_2}{\text{List } T_1 <: \text{List } T_2} \quad \text{S-LIST}$$

Definition 77 **Dict Subtyping** : Dictionaries are invariant in the key and covariant in the value (S-DICT).

$$\frac{K_1 \equiv K_2 \qquad V_1 <: V_2}{\text{Dict } K_1\ V_1 <: \text{Dict } K_2\ V_2} \quad \text{S-DICT}$$

Definition 78 **List Join** : Lists join pointwise in their element type (J-LIST).

$$\frac{}{\text{List } T_1 \sqcup \text{List } T_2 = \text{List } (T_1 \sqcup T_2)} \quad \text{J-LIST}$$

Definition 79 **Dict Join** : Dictionaries join only when keys are equivalent; the values join pointwise. If the keys differ, the result remains an explicit union (no structural alignment) (J-DICT).

$$\frac{K_1 \equiv K_2}{\text{Dict } K_1\ V_1 \sqcup \text{Dict } K_2\ V_2 = \text{Dict } K_1\ (V_1 \sqcup V_2)} \quad \text{J-DICT}$$

Definition 80 **List Meet** : Lists meet pointwise in their element type (M-LIST).

$$\frac{}{\text{List } T_1 \sqcap \text{List } T_2 = \text{List } (T_1 \sqcap T_2)} \quad \text{M-LIST}$$

Definition 81 **Dict Meet** : Dictionaries meet only when keys are equivalent; values meet pointwise. With unequal keys, the meet degenerates to a $\bot$ (no structural alignment) (M-DICT).

$$\frac{K_1 \equiv K_2}{\text{Dict } K_1\ V_1 \sqcap \text{Dict } K_2\ V_2 = \text{Dict } K_1\ (V_1 \sqcap V_2)} \quad \text{M-DICT}$$

Definition 82 **List Literal** : A list literal typechecks when its elements typecheck; the element type is the least upper bound (LUB) of all elements (T-LIST2). The empty list uses a fresh inference hole (Definition 19) for its element type (T-LIST1).

$$\frac{}{\Gamma \vdash [\,] : \text{List } \hat{\alpha}} \quad \text{T-LIST1}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \ldots \quad \Gamma \vdash e_n : T_n \quad \text{T} \coloneqq T_1 \sqcup \ldots \sqcup T_n}{\Gamma \vdash [e_1, \ldots, e_n] : \text{List T}} \quad \text{T-LIST2}$$

Definition 83 **List Indexing** : Lists are zero-indexed. Indexing a list by a subtype of an integer yields either the element type or `Null` (for negative or out-of-bounds indices) (T-SUBS).

Definition 84 **Dict Literal** : A dictionary literal typechecks when its entries typecheck: keys are invariant (all keys share one key type) and the dictionary's value type is the least upper bound (LUB) of the element value types (T-DICT2). The empty dictionary uses fresh inference holes (Definition 19) for both key and value types (T-DICT1).

$$\frac{}{\Gamma \vdash \{\}\colon \mathrm{Dict}\ \hat{\alpha}\ \hat{\beta}} \quad \text{T-DICT1}$$

$$\frac{\Gamma \vdash k_1\colon \mathrm{K} \quad \ldots \quad \Gamma \vdash k_n\colon \mathrm{K} \quad \Gamma \vdash v_1\colon V_1 \quad \ldots \quad \Gamma \vdash v_n\colon V_n \quad \mathrm{V} \coloneqq V_1 \sqcup \ldots \sqcup V_n}{\Gamma \vdash \{k_1 : v_1, \ldots, k_n : v_n\}\colon \mathrm{Dict\ K\ V}} \quad \text{T-DICT2}$$

Definition 85 **Dict Indexing** : Indexing a dictionary $Dict\ K\ V$ by a key of type K yields a value of V if the entry exists or $Null$ in its absence (T-SUBS).

Definition 86 **Type Subscript** : We model indexing at the type level by a partial function $\mathrm{sub}(T, K)$. It is defined for lists and dictionaries as:

$$\begin{gathered} \mathrm{sub} : H \times H \rightharpoonup H \\ \mathrm{dom}(\mathrm{sub}) \subseteq H \times H \\ \forall K : * .\forall V : * .\ \mathrm{sub}(Dict\ K\ V, K) = V \uplus Null \\ \forall I <: Integer.\forall T : * .\ \mathrm{sub}(List\ T, I) = T \uplus Null \end{gathered} \tag{20}$$

Outside $\mathrm{dom}(\mathrm{sub})$ the function is undefined.

Definition 87 **Term Subscript** : Indexing terms delegates to the type subscript (Definition 86): if $\mathrm{sub}(T, K)$ is defined, then $a[k]$ has type $\mathrm{sub}(T, K)$ (T-SUBS).

$$\frac{\Gamma \vdash \mathrm{a}\colon \mathrm{T} \quad \Gamma \vdash \mathrm{k}\colon \mathrm{K} \quad (T, K) \in \mathrm{dom}(\mathrm{sub})}{\Gamma \vdash \mathrm{a}[k]\colon \mathrm{sub}(T, K)} \quad \text{T-SUBS}$$

Implementation Note. In a future iteration of the compiler, indexing will return $Option\ T$ instead of $T \vee Null$. The core language currently lacks option-elimination constructs (e.g., pattern matching/combinators), so returning $Option\ T$ while possible would be unusable in practice. Once such forms exist, the rules will be revised to use $Option\ T$ (see Chapter 8).

6.9.6. Union

A union type $S \uplus T$ expresses alternatives: a term inhabits it iff it inhabits at least one summand. At the type level, $\uplus$ is a binary type connective corresponding to the lattice join (Definition 10); the join itself remains unmaterialised. We treat $\uplus$ as binary and right-associative, so $A \uplus B \uplus C$ abbreviates $A \uplus (B \uplus C)$, which keeps subtyping laws compositional.

Definition 88 **Union Constructor** : We introduce a binary constructor $\vec{\cup}$ for union types. The infix connective $\cup$ is definitionally equal to constructor application:

$$\vec{\cup}\, A\, B \triangleq A \cup B \tag{21}$$

The connective is binary and right-associative: $A \cup B \cup C \equiv A \cup (B \cup C)$.

Definition 89 **Union Kind** : Union is a binary type constructor. When applied to two term-classifying types, the result itself is term-classifying (K-UNION).

$$\frac{}{\Delta \vdash \vec{\cup} : * \Rightarrow * \Rightarrow *} \quad \text{K-UNION}$$

Definition 90 **Union Canonicalisation** : Unions are canonicalised under the lattice laws, viewing $\cup$ as the unmaterialised join: commutativity (U-COMM), associativity (U-ASSOC), idempotency (U-IDEM) and distribution over intersection (U-DIST).

$$\frac{}{\mathrm{S} \cup \mathrm{T} = \mathrm{T} \cup \mathrm{S}} \quad \text{U-COMM}$$

$$\frac{}{(S \cup T) \cup \mathrm{U} = \mathrm{S} \cup (T \cup U)} \quad \text{U-ASSOC}$$

$$\frac{}{\mathrm{T} \cup \mathrm{T} = \mathrm{T}} \quad \text{U-IDEM}$$

$$\frac{}{\mathrm{T} \cup (U \cap V) = (T \cup U) \cap (T \cup V)} \quad \text{U-DIST}$$

Definition 91 **Union Extrema** : For unions, $\bot$ is the identity (U-BOT) and $\top$ is absorbing (U-TOP).

$$\frac{}{\mathrm{T} \cup \bot = \mathrm{T}} \quad \text{U-BOT}$$

$$\frac{}{\mathrm{T} \cup \top = \top} \quad \text{U-TOP}$$

Definition 92 **Union Subtyping** : On the left, unions decompose (the rule is invertible): $(T \cup U) <: S$ iff $T <: S$ and $U <: S$ (S-UNION1). On the right, unions are introductions: $S <: (T \cup U)$ holds if $S <: T$ or $S <: U$ (S-UNION2, S-UNION3).

$$\frac{\mathrm{T} <: \mathrm{S} \qquad \mathrm{U} <: \mathrm{S}}{(T \cup U) <: \mathrm{S}} \quad \text{S-UNION1}$$

$$\frac{\mathrm{S} <: \mathrm{U}}{\mathrm{S} <: (T \cup U)} \quad \text{S-UNION2}$$

$$\frac{\mathrm{S} <: \mathrm{T}}{\mathrm{S} <: (T \cup U)} \quad \text{S-UNION3}$$

Lemma 93 **Union Left Inversion** : If $(T \uplus U) <: S$ then $T <: S \wedge U <: S$.

$$\dfrac{\dfrac{\dfrac{}{\mathrm{T} <: \mathrm{T}}\,\text{S-REFL} \quad \dfrac{\mathrm{S} <: \mathrm{T}}{\mathrm{S} <: (T \uplus U)}\,\text{S-UNION3}}{\mathrm{T} <: (T \uplus U)} \qquad (T \uplus U) <: \mathrm{S}}{\mathrm{T} <: \mathrm{S}}\,\text{S-TRANS} \quad \dfrac{\mathrm{T} <: \mathrm{U} \qquad \mathrm{U} <: \mathrm{V}}{\mathrm{T} <: \mathrm{V}}\,\text{U-INV1}$$

$$\dfrac{\dfrac{\dfrac{}{\mathrm{U} <: \mathrm{U}}\,\text{S-REFL} \quad \dfrac{\mathrm{S} <: \mathrm{U}}{\mathrm{S} <: (T \uplus U)}\,\text{S-UNION2}}{\mathrm{U} <: (T \uplus U)} \qquad (T \uplus U) <: \mathrm{S}}{\mathrm{U} <: \mathrm{S}}\,\text{S-TRANS} \quad \dfrac{\mathrm{T} <: \mathrm{U} \qquad \mathrm{U} <: \mathrm{V}}{\mathrm{T} <: \mathrm{V}}\,\text{U-INV2}$$

Lemma 94 **Union Right Inversion** : If $S <: (T \uplus U)$ then $S <: T \vee S <: U$.

Proof. We prove the statement by induction on the height of a derivation of $S <: (T \uplus U)$. In each case, we inspect the rule used at the root (outermost inference) of that derivation.

1. S-UNION2. The premise is $S <: T$, and the conclusion is $S <: (T \uplus U)$. Hence, the left disjunct $S <: T$ holds.
2. S-UNION3. The premise is $S <: U$, and the conclusion is $S <: (T \uplus U)$. Hence, the right disjunct $S <: U$ holds.
3. S-TRANS. We have premises $S <: V$ and $V <: (T \uplus U)$. By the induction hypothesis (IH) applied to the smaller derivation of $V <: (T \uplus U)$, either $V <: T \vee V <: U$. Compose with $S <: V$ via transitivity to obtain $S <: T \vee S <: U$.
4. *Exhaustion*. No other subtyping rule in the system has a conclusion whose right-hand side is a union. Thus the above cases are exhaustive.

Therefore, in all cases, $S <: T$ or $S <: U$. □

Lemma 95 **Union-Union Subtyping Decomposition** :

- *Binary*. $(S \uplus T) <: (U \uplus V)$ iff $(S <: U \vee S <: V) \wedge (T <: U \vee T <: V)$
- *n-ary* (finite). $\bigcup_{i=1}^{m} S_i <: \bigcup_{j=1}^{n} T_j$ iff $\bigwedge_{i=1}^{m}\left(\bigvee_{j=1}^{n} S_i <: T_j\right)$

Proof. (⇒) From $(S \uplus T) <: (U \uplus V)$ apply union-left inversion (Lemma 93) to get $S <: (U \uplus V) \wedge T <: (U \uplus V)$. Apply union-right inversion (Lemma 94) to each: $(S <: U \vee S <: V) \wedge (T <: U \vee T <: V)$

(⇐) Assume $(S <: U \vee S <: V) \wedge (T <: U \vee T <: V)$. By S-UNION2/S-UNION3 we obtain $(S <: (U \uplus V) \vee S <: (U \uplus V)) \wedge (T <: (U \uplus V) \vee T <: (U \uplus V))$. By propositional idempotence (meta-logic), each disjunction simplifies: $S <: (U \uplus V) \wedge T <: (U \uplus V)$. Use S-UNION1 to conclude $(S \uplus T) <: (U \uplus V)$ □

Since $\uplus$ is parsed right-associatively, the n-ary statement follows by repeated application of the binary case (equivalently, induction on m).

Lemma 96 **Union Monotonicity** : If $S_1 <: T_1$ and $S_2 <: T_2$, then $(S_1 \uplus S_2) <: (T_1 \uplus T_2)$

$$\dfrac{\dfrac{S_1 <: T_1 \quad \dfrac{\mathrm{S} <: \mathrm{T}}{\mathrm{S} <: (T \uplus U)}\,\text{S-UNION3}}{S_1 <: (T_1 \uplus T_2)} \qquad \dfrac{S_2 <: T_2 \quad \dfrac{\mathrm{S} <: \mathrm{U}}{\mathrm{S} <: (T \uplus U)}\,\text{S-UNION2}}{S_2 <: (T_1 \uplus T_2)} \qquad \dfrac{\mathrm{T} <: \mathrm{S} \quad \mathrm{U} <: \mathrm{S}}{(T \uplus U) <: \mathrm{S}}\,\text{S-UNION1}}{(S_1 \uplus S_2) <: (T_1 \uplus T_2)}\,\text{U-MONO}$$

Definition 97 **Union Meet** : Meet distributes over explicit unions (M-UNION).

$$\dfrac{}{(S \uplus T) \sqcap \mathrm{U} = (S \sqcap U) \uplus (T \sqcap U)}\,\text{M-UNION}$$

Definition 98 **Union Join** : Join distributes over explicit unions (J-UNION).

$$\dfrac{}{(S \uplus T) \sqcup \mathrm{U} = (S \sqcup U) \uplus (T \sqcup U)}\,\text{J-UNION}$$

Definition 99 **Union Introduction** : Any term of type T can be viewed as type $T \uplus U$ by subsumption, using reflexivity and right-introduction for unions (T-UNION).

$$\dfrac{\Gamma \vdash \mathrm{e}: \mathrm{T} \qquad \dfrac{\dfrac{}{\mathrm{T} <: \mathrm{T}}\,\text{S-REFL} \quad \dfrac{\mathrm{S} <: \mathrm{T}}{\mathrm{S} <: (T \uplus U)}\,\text{S-UNION3}}{\mathrm{T} <: (T \uplus U)} \qquad \dfrac{\Gamma \vdash \mathrm{e}: \mathrm{T} \quad \mathrm{T} <: \mathrm{U}}{\Gamma \vdash \mathrm{e}: \mathrm{U}}\,\text{T-SUB}}{\Gamma \vdash \mathrm{e}: \mathrm{T} \uplus \mathrm{U}}\,\text{T-UNION}$$

Definition 100 **Join Fallback** : If no non-fallback reduction applies to $T \sqcup U$, we materialise an explicit union (J-FALL). This guarantees the totality of lattice evaluation.

$$\dfrac{\nexists\, \mathrm{S}.\ \mathrm{T} \sqcup \mathrm{U} \Downarrow \mathrm{S}}{\mathrm{T} \sqcup \mathrm{U} = \mathrm{T} \uplus \mathrm{U}}\,\text{J-FALL}$$

Definition 101 **Union Projection** : We extend the partial function proj (Definition 47) pointwise over unions, defined only when each disjunct admits the same key:

$$\forall (S_1, \ell) \in \mathrm{dom}(\mathrm{proj}).\forall (S_2, \ell) \in \mathrm{dom}(\mathrm{proj}).\ \mathrm{proj}(S_1 \uplus S_2, \ell) = \mathrm{proj}(S_1, \ell) \sqcup \mathrm{proj}(S_2, \ell) \quad (22)$$

Definition 102 **Union Subscript** : We extend the partial function sub (Definition 86) pointwise over unions, defined only when each disjunct admits the same key:

$$\forall (S_1, K) \in \mathrm{dom}(\mathrm{sub}).\forall (S_2, K) \in \mathrm{dom}(\mathrm{sub}).\ \mathrm{sub}(S_1 \uplus S_2, K) = \mathrm{sub}(S_1, K) \sqcup \mathrm{sub}(S_2, K) \quad (23)$$

6.9.7. Intersection

An intersection type $A \curlywedge T$ expresses conjunction: a term inhabits it iff it inhabits both summands. At the type level, $\curlywedge$ is a binary connective corresponding to the lattice meet (Definition 9); the meet itself remains unmaterialised. We treat $\curlywedge$ as binary and right-associative, so $A \curlywedge B \curlywedge C$ abbreviates $A \curlywedge (B \curlywedge C)$, which keeps subtyping laws compositional.

Definition 103 **Intersection Constructor** : We introduce a binary constructor $\vec{\cap}$ for intersection types. The infix connective $\cap$ is definitionally equal to constructor application:

$$\vec{\cap}\, A\, B \triangleq A \cap B \tag{24}$$

The connective is binary and right-associative: $A \cap B \cap C \equiv A \cap (B \cap C)$.

Definition 104 **Intersection Kind** : Intersections are binary type constructors. When applied to two term-classifying types, the result itself is term-classifying (K-INTER).

$$\frac{}{\Delta \vdash \vec{\cap} : * \Rightarrow * \Rightarrow *} \text{K-INTER}$$

Definition 105 **Intersection Canonicalisation** : Intersections are canonicalised under the lattice laws, viewing $\cap$ as the unmaterialised meet: commutativity (I-COMM), associativity (I-ASSOC), idempotency (I-IDEM) and distribution over union (I-DIST).

$$\frac{}{\mathrm{S} \cap \mathrm{T} = \mathrm{T} \cap \mathrm{S}} \text{I-COMM}$$

$$\frac{}{(S \cap T) \cap \mathrm{U} = \mathrm{S} \cap (T \cap U)} \text{I-ASSOC}$$

$$\frac{}{\mathrm{T} \cap \mathrm{T} = \mathrm{T}} \text{I-IDEM}$$

$$\frac{}{\mathrm{T} \cap (U \uplus V) = (T \cap U) \uplus (T \cap V)} \text{I-DIST}$$

Definition 106 **Intersection Extrema** : For intersections, $\top$ is the identity (I-TOP) and $\bot$ is absorbing (I-BOT).

$$\frac{}{\mathrm{T} \cap \top = \mathrm{T}} \text{I-TOP}$$

$$\frac{}{\mathrm{T} \cap \bot = \bot} \text{I-BOT}$$

Definition 107 **Intersection Subtyping** : On the left, the intersections project: $(T \cap U) <: T$ and $(T \cap U) <: U$ (S-INTER1, S-INTER2). On the right, intersections are (invertible) introductions: $S <: (T \cap U)$ holds iff $S <: T$ and $S <: U$ (S-INTER3)

$$\frac{}{(T \cap U) <: \mathrm{T}} \text{S-INTER1}$$

$$\frac{}{(T \cap U) <: \mathrm{U}} \text{S-INTER2}$$

$$\frac{\mathrm{S} <: \mathrm{T} \qquad \mathrm{S} <: \mathrm{U}}{\mathrm{S} <: (T \cap U)} \text{S-INTER3}$$

Lemma 108 **Intersection Right Inversion** : If $S <: (T \cap U)$ then $S <: T$ and $S <: U$.

$$\dfrac{\mathrm{S} <: (T \cap U) \qquad \dfrac{}{(T \cap U) <: \mathrm{T}}\,\text{S-INTER1} \qquad \dfrac{\mathrm{T} <: \mathrm{U} \qquad \mathrm{U} <: \mathrm{V}}{\mathrm{T} <: \mathrm{V}}\,\text{S-TRANS}}{\mathrm{S} <: \mathrm{T}}\,\text{I-INV1}$$

$$\dfrac{\mathrm{S} <: (T \cap U) \qquad \dfrac{}{(T \cap U) <: \mathrm{U}}\,\text{S-INTER2} \qquad \dfrac{\mathrm{T} <: \mathrm{U} \qquad \mathrm{U} <: \mathrm{V}}{\mathrm{T} <: \mathrm{V}}\,\text{S-TRANS}}{\mathrm{S} <: \mathrm{U}}\,\text{I-INV2}$$

Lemma 109 **Intersection Monotonicity** : If $S_1 <: T_1$ and $S_2 <: T_2$, then $(S_1 \cap S_2) <: (T_1 \cap T_2)$.

$$\dfrac{\dfrac{\dfrac{}{(T \cap U) <: \mathrm{T}}\,\text{S-INTER1} \qquad S_1 <: T_1}{(S_1 \cap S_2) <: T_1} \qquad \dfrac{\dfrac{\dfrac{}{(T \cap U) <: \mathrm{U}}\,\text{S-INTER2} \qquad S_2 <: T_2}{(S_1 \cap S_2) <: T_2} \qquad \dfrac{\mathrm{S} <: \mathrm{T} \qquad \mathrm{S} <: \mathrm{U}}{\mathrm{S} <: (T \cap U)}\,\text{S-INTER3}}{(S_1 \cap S_2) <: (T_1 \cap T_2)}\,\text{I-MONO}$$

Definition 110 **Intersection Meet** : Meet distributes over explicit intersections (J-UNION).

$$\dfrac{}{(S \cap T) \sqcap \mathrm{U} = (S \sqcap U) \cap (T \sqcap U)}\,\text{M-INTER}$$

Definition 111 **Intersection Join** : Join distributes over explicit intersections (J-INTER).

$$\dfrac{}{(S \cap T) \sqcup \mathrm{U} = (S \sqcup U) \cap (T \sqcup U)}\,\text{J-INTER}$$

Definition 112 **Intersection Introduction** : A term inhabits an intersection exactly when it independently inhabits both components (T-INTER1).

$$\dfrac{\Gamma \vdash \mathrm{e}: \mathrm{T} \qquad \Gamma \vdash \mathrm{e}: \mathrm{U}}{\Gamma \vdash \mathrm{e}: \mathrm{T} \cap \mathrm{U}}\,\text{T-INTER1}$$

Definition 113 **Intersection Elimination** : From a term of type $T \cap U$ we may project either component via subsumption and left-projection (T-INTER2, T-INTER3).

$$\dfrac{\Gamma \vdash \mathrm{e}: \mathrm{T} \cap \mathrm{U} \qquad \dfrac{}{(T \cap U) <: \mathrm{T}}\,\text{S-INTER1} \qquad \dfrac{\Gamma \vdash \mathrm{e}: \mathrm{T} \qquad \mathrm{T} <: \mathrm{U}}{\Gamma \vdash \mathrm{e}: \mathrm{U}}\,\text{T-SUB}}{\Gamma \vdash \mathrm{e}: \mathrm{T}}\,\text{T-INTER2}$$

$$\dfrac{\Gamma \vdash \mathrm{e}: \mathrm{T} \cap \mathrm{U} \qquad \dfrac{}{(T \cap U) <: \mathrm{U}}\,\text{S-INTER2} \qquad \dfrac{\Gamma \vdash \mathrm{e}: \mathrm{T} \qquad \mathrm{T} <: \mathrm{U}}{\Gamma \vdash \mathrm{e}: \mathrm{U}}\,\text{T-SUB}}{\Gamma \vdash \mathrm{e}: \mathrm{U}}\,\text{T-INTER3}$$

Definition 114 **Meet Fallback** : If no non-fallback reduction applies to $T \sqcap U$, we materialise an explicit intersection (M-FALL). This guarantees the totality of lattice evaluation.

$$\frac{\nexists\, S.\ T \sqcap U \Downarrow_0 S}{T \sqcap U = T \frown U} \qquad \text{M-FALL}$$

Definition 115 **Intersection Projection** : We extend the partial function proj (Definition 47) pointwise over intersections, defined only when each conjunct admits the same key:

$$\forall (S_1, \ell) \in \mathrm{dom}(\mathrm{proj}). \forall (S_2, \ell) \in \mathrm{dom}(\mathrm{proj}).\ \mathrm{proj}(S_1 \frown S_2, \ell) = \mathrm{proj}(S_1, \ell) \sqcap \mathrm{proj}(S_2, \ell) \quad (25)$$

Definition 116 **Intersection Subscript** : We extend the partial function sub (Definition 86) pointwise over intersections, defined only when each conjunct admits the same key:

$$\forall (S_1, K) \in \mathrm{dom}(\mathrm{sub}). \forall (S_2, K) \in \mathrm{dom}(\mathrm{sub}).\ \mathrm{sub}(S_1 \frown S_2, K) = \mathrm{sub}(S_1, K) \sqcap \mathrm{sub}(S_2, K) \quad (26)$$

6.10. Inference

We present a constraint-based inference algorithm for HashQL. It builds on the Hindley–Milner (HM) type system [36] and Algorithm W [18], and follows the HM(X) separation between a syntax-directed constraint generator and an independent solver [52].

HashQL departs from plain HM through subtyping, explicit unions/intersections, nominal/structural mixing, and variance; as is typical for such extensions, we do not claim principal types or completeness across the entire language.

Constraints are interpreted over the lattice in Chapter 6.4; the solver relies on variance-aware subtyping (Definition 6), meet (Definition 9), and join (Definition 10), with principled constraint simplification in the style of François Pottier. [41] and François Pottier. [42].

6.10.1. Subjects

The solver ranges over a fixed, disjoint supply of subject variables; for any given program, we restrict to the finite subset that occurs: $\mathcal{V} \subseteq \mathcal{B} \uplus \mathcal{U}$, where

- $\mathcal{B} = \{\alpha, \beta, \gamma, \ldots\}$ are binder-introduced variables (Definition 16, Definition 17),
- $\mathcal{U} = \{\hat{\alpha}, \hat{\beta}, \hat{\gamma}, \ldots\}$ are inference holes (Definition 19).

All subjects are term-classifying:

$$\forall v \in \mathcal{V}. \Delta \vdash v : * \quad (27)$$

Definition 117 **Origin** : Each variable carries an origin tag:

$$\begin{aligned} &\mathrm{origin} : \mathcal{V} \to \{\mathrm{hole}, \mathrm{bound}\} \\ &\mathrm{origin}(v) = \begin{cases} \mathrm{bound} & \text{if } v \in \mathcal{B} \\ \mathrm{hole} & \text{if } v \in \mathcal{U} \end{cases} \end{aligned} \tag{28}$$

Equip $\{\mathrm{hole}, \mathrm{bound}\}$ with the preorder $\mathrm{hole} \leq \mathrm{bound}$. We write $\sqcup$ and $\sqcap$ for its join (supremum) and meet (infimum), respectively.

Definition 118 **Subject Classes** : Let $\approx$ be the smallest equivalence relation on $\mathcal{V}$ closed under explicit equalities (and unifications), and such that if $X \leq Y$ and $Y \leq X$ hold, then $X \approx Y$.

We write $[v]$ for the class of v.

Let $\mathcal{K}$ be the set of equivalence classes:

$$\begin{aligned} \mathcal{K} &\coloneqq \{[v] \mid v \in \mathcal{V}\} \\ [v] &\coloneqq \{u \in \mathcal{V} \mid u \approx v\} \end{aligned} \tag{29}$$

The mode of a class is the join of its members' origins:

$$\begin{aligned} &\mathrm{mode} : \mathcal{K} \to \{\mathrm{hole}, \mathrm{bound}\} \\ &\mathrm{mode}([v]) \coloneqq \bigsqcup_{u \in [v]} \mathrm{origin}(u) \end{aligned} \tag{30}$$

Definition 119 **Subject Representatives** : A representative is a choice function on equivalence classes:

$$\mathrm{rep} : \mathcal{K} \to \mathcal{V}, \quad \mathrm{rep}([v]) \in [v]. \tag{31}$$

We write $\bar{v} \coloneqq \mathrm{rep}([v])$ ("the representative of v").

The representative must be:

- Well-defined: $\forall u, v \in \mathcal{V}. u \approx v \Rightarrow \bar{u} = \bar{v}$
- Idempotent: $\forall v \in \mathcal{V}. \bar{\bar{v}} = \bar{v}$

The choice of rep is implementation-defined. It is fixed per solver round and remains unchanged as long as the partition induced by $\approx$ does not change.

During solving, variables are handled uniformly. At finalisation, classes are treated according to their mode:

- If $\mathrm{mode}([v]) = \mathrm{bound}$, no witness type is required (the class may remain abstract).
- If $\mathrm{mode}([v]) = \mathrm{hole}$, an unsolved class is a type error.

6.10.2. Constraints

Before solving, we generate a finite constraint set Ξ from all top-level subtyping obligations. For a finite $\mathcal{O} \subseteq H \times H$,

$$\Xi \coloneqq \bigcup_{(S,T)\in\mathcal{O}} \mathrm{col}(S,T). \tag{32}$$

We use two (pure) functions that each return a finite set of constraints:

- $\mathrm{col}(S,T)$ – collect constraints for the obligation $S <: T$;
- $\mathrm{dep}(\alpha,T)$ – add structural dependencies from α to variables in positive positions inside T (invoked by col immediately after emitting $\alpha \le T$).

Constraints are triples $(O,L,R) \in \mathcal{R}\times\mathcal{V}\times\mathcal{X}$ (operator, subject, target) with

$$\begin{aligned} \mathrm{Key} &\coloneqq \mathrm{Ident} \uplus \mathbb{Z} \\ \mathrm{Select} &\coloneqq (\mathrm{proj}, H, \mathrm{Key}) \mid (\mathrm{sub}, H, H) \\ \mathcal{R} &\coloneqq \{\mathtt{leq}, \mathtt{geq}, \mathtt{eq}, \mathtt{dep}, \mathtt{unify}, \mathtt{ord}, \mathtt{sel}\} \\ \mathcal{X} &\coloneqq H \mid \mathcal{V} \mid \mathrm{Select}. \end{aligned} \tag{33}$$

Valid atoms (always with the subject variable in the middle slot L):

- Upper bound $\alpha \le T$: $(\mathtt{leq}, \alpha, T)$
- Lower bound $T \le \alpha$: $(\mathtt{geq}, \alpha, T)$
- Rigid equality $\alpha \equiv T$: $(\mathtt{eq}, \alpha, T)$
- Variable ordering $\alpha \le \beta$: $(\mathtt{ord}, \alpha, \beta)$
- Variable unification $\alpha \equiv \beta$: $(\mathtt{unify}, \alpha, \beta)$
- Dependency $\alpha \dashrightarrow \beta$: $(\mathtt{dep}, \alpha, \beta)$
- Projection $\hat{\alpha} \coloneqq \mathrm{proj}(T,\ell)$: $(\mathtt{sel}, \hat{\alpha}, (\mathrm{proj}, T, \ell))$
- Indexing $\hat{\alpha} \coloneqq \mathrm{sub}(T,K)$: $(\mathtt{sel}, \hat{\alpha}, (\mathrm{sub}, T, K))$

All atoms except projection and indexing are discharged immediately inside col. Projection and indexing are emitted as equalities to the totalised operators below (Definition 120, Definition 121) and discharged at the term sites that introduced them.

We write $\Xi\ \mathrm{wf}$ as a well-formedness side condition: every element of Ξ is one of the atoms above, and its right-hand operand has the corresponding shape.

Definition 120 **Total Projection** $\mathrm{proj}_\forall$: $\mathrm{proj}_\forall$ totalises proj (Definition 47):

$$\begin{aligned} \mathrm{proj}_\forall &: H \times \mathrm{Key} \to H, \\ \mathrm{proj}_\forall(T,\ell) &\coloneqq \begin{cases} \mathrm{proj}(T,\ell) & \text{if } (T,\ell) \in \mathrm{dom}(\mathrm{proj}) \\ \bot & \text{otherwise} \end{cases} \end{aligned} \tag{34}$$

Definition 121 **Total Subscript** $\mathrm{sub}_\forall$: $\mathrm{sub}_\forall$ totalises sub (Definition 86):

$$\begin{aligned} \mathrm{sub}_\forall &: H \times H \to H, \\ \mathrm{sub}_\forall(T,K) &\coloneqq \begin{cases} \mathrm{sub}(T,K) & \text{if } (T,K) \in \mathrm{dom}(\mathrm{sub}) \\ \bot & \text{otherwise} \end{cases} \end{aligned} \tag{35}$$

Definition 122 **Subject Occurrence** occ : We collect variables that appear as subjects, and also those that appear as entire right–hand sides when the right-hand side is a variable (as is the case for the ord and unify operators):

$$\begin{aligned} &\mathrm{occ} : \mathcal{P}(\mathcal{R} \times \mathcal{V} \times \mathcal{X}) \to \mathcal{P}(\mathcal{V}) \\ &\mathrm{occ}(\Xi) = \bigcup_{(O,L,R)\in\Xi} \{L\} \cup \begin{cases} \{R\} \text{ if } R \in \mathcal{V} \\ \varnothing \quad \text{otherwise} \end{cases} \end{aligned} \tag{36}$$

6.10.3. Constraint Collection

We introduce a total function col to generate (sound) constraint atoms from a subtyping obligation. It decomposes structurally when available and otherwise emits variable–centred atoms. A final subtyping check is performed after solving; generation may therefore be liberal, provided every emitted atom is a logical consequence of $S <: T$.

We define three functions:

- variance-aware col – entrypoint, using Φ (Chapter 6.7);
- declarative $\mathrm{col}_{\leq}$ – for $\leq$-positions (Definition 5);
- declarative $\mathrm{col}_{\equiv}$ – for invariant positions (Definition 7).

To enable more ergonomic notation during projection, we define several auxiliary functions:

$$\begin{aligned} &\mathrm{dom}_{\cap}(S,T) = \mathrm{dom}(S) \cap \mathrm{dom}(T) \\ &\mathrm{col}_{\mathrm{proj}} : H \times H \times \mathrm{Key} \rightharpoonup \mathcal{P}(\mathcal{R} \times \mathcal{V} \times \mathcal{X}) \\ &\forall (S,\ell) \in \mathrm{dom}(\mathrm{proj}). \\ &\quad \forall (T,\ell) \in \mathrm{dom}(\mathrm{proj}). \\ &\qquad \mathrm{col}_{\mathrm{proj}}(S,T,\ell) = \mathrm{col}(\mathrm{proj}(S,\ell), \mathrm{proj}(T,\ell)) \end{aligned} \tag{37}$$

Variance-aware entrypoint. Mirror Definition 35: polarise the obligation with Φ and dispatch to the appropriate declarative collector depending on the variance:

$$\begin{aligned} &\mathrm{col} : H \times H \to \mathcal{P}(\mathcal{R} \times \mathcal{V} \times \mathcal{X}) \\ &\mathrm{col}(S,T) = \begin{cases} \Phi, \varphi : \mathrm{trans}_{\Phi}(S,T) \vdash \mathrm{col}_{\leq}(S,T) \text{ if } \mathrm{flow}_{\Phi}(S,T) = +1 \\ \Phi, \varphi : \mathrm{trans}_{\Phi}(S,T) \vdash \mathrm{col}_{\leq}(T,S) \text{ if } \mathrm{flow}_{\Phi}(S,T) = -1 \\ \Phi, \varphi : \mathrm{trans}_{\Phi}(S,T) \vdash \mathrm{col}_{\equiv}(S,T) \text{ otherwise} \end{cases} \end{aligned} \tag{38}$$

Declarative subtyping collector. Decompose structurally when both sides share shape; otherwise, emit variable-centred atoms:

$$\mathrm{col}_{\leq} : H \times H \to \mathcal{P}(\mathcal{R} \times \mathcal{V} \times \mathcal{X})$$

$$\mathrm{col}_{\leq}(S,T) = \begin{cases} \bigcup_{\ell \in \mathrm{dom}_{\cap}(S,T)} \mathrm{col}_{\mathrm{proj}}(S,T,\ell) & \text{if } S = (\omega \mid \ldots) \wedge T = (\omega \mid \ldots) \\ \bigcup_{i \in \mathrm{dom}_{\cap}(S,T)} \mathrm{col}_{\mathrm{proj}}(S,T,i) & \text{if } S = A_0 \times \ldots \times A_{n-1} \wedge T = B_0 \times \ldots \times B_{n-1} \\ \mathrm{col}(A_2,A_1) \cup \mathrm{col}(B_1,B_2) & \text{if } S = A_1 \to B_1 \wedge T = A_2 \to B_2 \\ \mathrm{col}_{\equiv}(K_1,K_2) \cup \mathrm{col}(V_1,V_2) & \text{if } S = \mathit{Dict}\ K_1 V_1 \wedge T = \mathit{Dict}\ K_2 V_2 \\ \mathrm{col}(A_1,A_2) & \text{if } S = \mathit{List}\ A_1 \wedge T = \mathit{List}\ A_2 \\ \mathrm{col}(S_1,T) \cup \mathrm{col}(S_2,T) & \text{if } S = S_1 \uplus S_2 \\ \mathrm{col}(S_1,T) \cup \mathrm{col}(S_2,T) & \text{if } S = S_1 \cap S_2 \\ \mathrm{col}(S,T_1) \cup \mathrm{col}(S,T_2) & \text{if } T = T_1 \cap T_2 \\ \{(\mathtt{ord},S,T)\} & \text{if } S \in \mathcal{V} \wedge T \in \mathcal{V} \\ \{(\mathtt{leq},S,T)\} \cup \mathrm{dep}(S,+1,T) & \text{if } S \in \mathcal{V} \\ \{(\mathtt{geq},T,S)\} \cup \mathrm{dep}(T,-1,S) & \text{if } T \in \mathcal{V} \\ \varnothing & \text{if } T = T_1 \uplus T_2 \\ \ldots & \\ \varnothing & \text{otherwise} \end{cases} \tag{39}$$

Note. $\mathrm{col}_{\leq}$ does not descend on the right through a union. Such a descent would produce disjunctive obligations, and the solver admits only conjunctive constraint sets; the remaining choice is deferred to the final subtyping check.

Declarative equality collector. Enforce definitional equality as mutual subtyping: generate obligations for both directions and decompose structurally:

$$\mathrm{col}_{\equiv} : H \times H \to \mathcal{P}(\mathcal{R} \times \mathcal{V} \times \mathcal{X})$$

$$\mathrm{col}_{\equiv}(S,T) = \begin{cases} \bigcup_{\ell \in \mathrm{dom}_{\cap}(S,T)} \mathrm{col}_{\mathrm{proj}}(S,T,\ell) & \text{if } S = (\omega \mid \ldots) \wedge T = (\omega \mid \ldots) \\ \bigcup_{i \in \mathrm{dom}_{\cap}(S,T)} \mathrm{col}_{\mathrm{proj}}(S,T,i) & \text{if } S = A_0 \times \ldots \times A_{n-1} \wedge T = B_0 \times \ldots \times B_{n-1} \\ \mathrm{col}(A_1,A_2) \cup \mathrm{col}(B_1,B_2) & \text{if } S = A_1 \to B_1 \wedge T = A_2 \to B_2 \\ \mathrm{col}(K_1,K_2) \cup \mathrm{col}(V_1,V_2) & \text{if } S = \mathit{Dict}\ K_1 V_1 \wedge T = \mathit{Dict}\ K_2 V_2 \\ \mathrm{col}(A_1,A_2) & \text{if } S = \mathit{List}\ A_1 \wedge T = \mathit{List}\ A_2 \\ \mathrm{col}(S_1,T) \cup \mathrm{col}(S_2,T) & \text{if } S = S_1 \cap S_2 \\ \mathrm{col}(S,T_1) \cup \mathrm{col}(S,T_2) & \text{if } T = T_1 \cap T_2 \\ \{(\mathtt{unify},S,T)\} & \text{if } S \in \mathcal{V} \wedge T \in \mathcal{V} \\ \{(\mathtt{eq},S,T)\} & \text{if } S \in \mathcal{V} \\ \{(\mathtt{eq},T,S)\} & \text{if } T \in \mathcal{V} \\ \varnothing & \text{if } S = S_1 \uplus S_2 \\ \varnothing & \text{if } T = T_1 \uplus T_2 \\ \ldots & \\ \varnothing & \text{otherwise} \end{cases} \tag{40}$$

Note. $\mathrm{col}_{\equiv}$ never descends through unions. Since $\equiv$ is definitionally mutual subtyping (Definition 7), any descent would recreate the union-right case in $\mathrm{col}_{\leq}$, which we intentionally suppress.

Note. We do not enumerate transparent wrappers – μ-types, nominal wrappers, generic abstraction/quantification, type application, and substitution – in the clauses of $\mathrm{col}_{\leq}$ and $\mathrm{col}_{\equiv}$. Each merely forwards collection to its body/arguments and contributes no constraints beyond those of its children. Writing them out would duplicate rules without changing the generated constraint set. Hence, throughout, we treat such wrappers as transparent: collection proceeds underneath them as if the wrapper were not present.

6.10.4. Dependency Collection

We introduce a total function dep to generate structural dependencies between variables and are variance agnostic. A dependency records that the choice for one variable must be known before finalising another. These edges are emitted by col immediately after producing $(\mathtt{leq}, \alpha, T)$ or $(\mathtt{geq}, \alpha, T)$.

$$\mathrm{dep} : \mathcal{V} \times \{+1, -1\} \times H \to \mathcal{P}(\mathcal{R} \times \mathcal{V} \times \mathcal{X})$$

$$\mathrm{dep}(\alpha, \tau, T) = \begin{cases} \bigcup_{\ell \in \mathrm{dom}(T)} \mathrm{dep}(\alpha, \tau, \mathrm{proj}(T, \ell)) & \text{if } T = (\omega \mid \ldots) \\ \bigcup_{i \in \mathrm{dom}(T)} \mathrm{dep}(\alpha, \tau, \mathrm{proj}(T, i)) & \text{if } T = A_0 \times \ldots \times A_{n-1} \\ \mathrm{dep}(\alpha, \tau, A) \cup \mathrm{dep}(\alpha, \tau, B) & \text{if } T = A \to B \\ \mathrm{dep}(\alpha, \tau, K) \cup \mathrm{dep}(\alpha, \tau, V) & \text{if } T = \mathit{Dict}\ K\ V \\ \mathrm{dep}(\alpha, \tau, A) & \text{if } T = \mathit{List}\ A \\ \mathrm{dep}(\alpha, \tau, T_1) \cup \mathrm{dep}(\alpha, \tau, T_2) & \text{if } T = T_1 \uplus T_2 \\ \mathrm{dep}(\alpha, \tau, T_1) \cup \mathrm{dep}(\alpha, \tau, T_2) & \text{if } T = T_1 \cap T_2 \\ \mathrm{dep}(\alpha, \tau, A) & \text{if } T = o\langle s | A \rangle \\ \mathrm{dep}(\alpha, \tau, A) & \text{if } T = \mu\beta.B \\ \mathrm{dep}(\alpha, \tau, A) \cup \mathrm{dep}(\alpha, \tau, B) & \text{if } T = AB \\ \mathrm{dep}(\alpha, \tau, A) \cup \mathrm{dep}(\alpha, \tau, B) & \text{if } T = A[\gamma \mapsto B] \\ \mathrm{dep}(\alpha, \tau, B) & \text{if } T = \Lambda\gamma : K.B \\ \mathrm{dep}(\alpha, \tau, B) & \text{if } T = \forall\gamma.B \\ \{(\mathrm{dep}, \alpha, T)\} & \text{if } T \in \mathcal{V} \wedge \tau = +1 \\ \{(\mathrm{dep}, T, \alpha)\} & \text{if } T \in \mathcal{V} \wedge \tau = -1 \\ \varnothing & \text{otherwise} \end{cases} \tag{41}$$

6.10.5. Solver

We solve a finite constraint set Ξ to produce a substitution $\upsilon : \mathcal{V} \to H$. The solver operates over the type lattice (Chapter 6.4), iterates to a fixpoint, and never invents new types.

The algorithm runs as a fixpoint pipeline; each phase may add new constraints to Ξ as facts are uncovered:

- *Unification* (Chapter 6.10.5.1): enforce antisymmetry and form equivalence classes.
- *Normalisation* (Chapter 6.10.5.2): normalise the environment to enable later phases.
- *Constraint propagation* (Chapter 6.10.5.3): propagate lower/upper bounds (forward/backward) along the constraint graph.
- *Validation* (Chapter 6.10.5.4): check bound/equality compatibility and decide witnesses.
- *Selection* (Chapter 6.10.5.5): resolve deferred projections/subscripts.

Phases repeat until no new constraints or resolutions appear. At convergence, the solver simplifies υ pointwise under υ.

6.10.5.1. Unification

Unification identifies variables that must denote the same type. It collapses cycles in the variable–variable order and respects explicit variable equalities. The step only coarsens

the partition, but never introduces new variables or splits classes. The concrete strongly-connected components (SCC) algorithm is implementation-defined.

Definition 123 **Unification Operator** U : Build a directed graph G over the representatives of each variable:

$$\begin{aligned}
\text{edge}(\alpha,\beta) &\coloneqq (\bar{\alpha},\bar{\beta}) \\
N &\coloneqq \{\bar{v} \mid v \in \text{occ}(\Xi) \cap \mathcal{V}\} \\
E_0 &\coloneqq \{\text{edge}(\alpha,\beta) \mid (\texttt{ord},\alpha,\beta) \in \Xi\} \\
&\quad \cup \{\text{edge}(\alpha,\beta) \mid (\texttt{unify},\alpha,\beta) \in \Xi\} \\
&\quad \cup \{\text{edge}(\beta,\alpha) \mid (\texttt{unify},\alpha,\beta) \in \Xi\} \\
&\quad \cup \{\text{edge}(\alpha,\beta) \mid (\texttt{eq},\alpha,\beta) \in \Xi \wedge \beta \in \mathcal{V}\} \\
&\quad \cup \{\text{edge}(\beta,\alpha) \mid (\texttt{eq},\alpha,\beta) \in \Xi \wedge \beta \in \mathcal{V}\} \\
E &\coloneqq E_0 \setminus \{(r,r) \mid r \in N\} \\
G &\coloneqq (N,E)
\end{aligned} \tag{42}$$

Let $\text{SCC}(G)$ be the set of SCC of G (implementation defined):

$$\text{SCC} : (N,E) \to \mathcal{P}(\mathcal{P}(N)) \tag{43}$$

Merge the equivalence classes along the SCC:

$$\begin{aligned}
\mathcal{K}' &\coloneqq \left\{ \bigcup_{r \in C} [r] \;\middle|\; C \in \text{SCC}(G) \right\} \\
\approx' &\coloneqq \bigcup_{c \in \mathcal{K}'} c \times c
\end{aligned} \tag{44}$$

Choose any representative function rep' for $\approx'$ (Definition 119) and return $(\approx', \text{rep}')$. The constraint set Ξ itself is not rewritten here; normalisation (Chapter 6.10.5.2) performs the rewrite using rep'.

By construction U is:

- Monotone: the partition coarsens ($|\mathcal{K}'| \leq |\mathcal{K}|$).
- Idempotent (fixed Ξ): a second application with the same Ξ leaves $\approx'$ unchanged.

Lemma 124 **Unification Soundness** : Let U be the unification step defined above, and let $\approx'$ be the updated equivalence after applying U to Ξ. Let rep' be any representative function for $\approx'$ with $\mathrm{rep}'([v]_{\approx'}) \in [v]_{\approx'}$ and write $\tilde{v} \coloneqq \mathrm{rep}'([v]_{\approx'})$.

Define the representative-rewritten constraint set:

$$\Xi' \coloneqq \left\{ \left(O, \tilde{L}, \begin{cases} \tilde{R} \text{ if } R \in \mathcal{V} \\ R \text{ otherwise} \end{cases} \right) \middle| (O, L, R) \in \Xi \right\} \tag{45}$$

Then for any substitution v over $\mathcal{V}$, v satisfies Ξ iff v satisfies Ξ' (up to definitional equality of types).

Proof. ($\Rightarrow$) If v satisfies Ξ, unification only merges variables that are mutually constrained ($x \leq y$ and $y \leq x$) or explicitly equated. Hence, all members of any $[v]_{\approx'}$ are equal in all models of Ξ. Replacing L by $\tilde{L}$ (and R by $\tilde{R}$ when $R \in \mathcal{V}$) preserves the truth of each atom, so v satisfies Ξ'.

($\Leftarrow$) Conversely, atoms of Ξ are identical to (or a variable-renaming of) atoms in Ξ'. Renaming to an $\approx'$-equivalent variable preserves satisfaction. Thus if v satisfies Ξ', it satisfies Ξ. □

6.10.5.2. Normalisation

Normalisation prepares the constraint set for propagation by :

- canonicalising all types using the lattice laws
- rewriting variables to their current representatives
- eliminating tautologies.

It does not decide subtyping, does not resolve selections, and does not invent new types.

Definition 125 **Normalisation Operator** N : Normalisation is a semantics-preserving preprocessing step. It replaces variables by their class representatives and removes vacuous constraint atoms (Chapter 6.10.2). It does not appeal to semantic subtyping beyond recognising $\top/\bot$ and does not strengthen constraints; satisfiability and the solution set are preserved.

We define a total function norm which canonicalises one constraint by mapping the subject to its representative (and the right–hand side only when it is itself a variable). Structure on the right–hand side is left unchanged.

$$\begin{aligned} \mathrm{norm} &: \mathcal{R} \times \mathcal{V} \times \mathcal{X} \to \mathcal{R} \times \mathcal{V} \times \mathcal{X} \\ \mathrm{norm}(O, L, R) &= \left(O, \bar{L}, \begin{cases} \bar{R} \text{ if } R \in \mathcal{V} \\ R \text{ otherwise} \end{cases} \right) \end{aligned} \tag{46}$$

We define triv on a finite set Ξ to return exactly the tuples that cannot constrain any solution (vacuous atoms). Trivial constraints can be safely omitted.

$$\begin{aligned} \mathrm{triv} &: \mathcal{P}(\mathcal{R} \times \mathcal{V} \times \mathcal{X}) \to \mathcal{P}(\mathcal{R} \times \mathcal{V} \times \mathcal{X}) \\ \mathrm{triv}(\Xi) &= \{(O, L, R) \in \Xi \mid \mathrm{trivial}(O, L, R)\} \end{aligned}$$

$$\begin{aligned} \mathrm{trivial} &: \mathcal{R} \times \mathcal{V} \times \mathcal{X} \to \{0, 1\} \\ \mathrm{trivial}(O, L, R) &\coloneqq (O = \texttt{leq} \wedge R = \top) \\ &\quad \vee (O = \texttt{geq} \wedge R = \bot) \\ &\quad \vee \left(O = \texttt{ord} \wedge \bar{L} = \bar{R}\right) \\ &\quad \vee \left(O = \texttt{dep} \wedge \bar{L} = \bar{R}\right) \\ &\quad \vee \left(O = \texttt{unify} \wedge \bar{L} = \bar{R}\right) \\ &\quad \vee \left(O = \texttt{eq} \wedge R \in \mathcal{V} \wedge \bar{L} = \bar{R}\right) \end{aligned} \tag{47}$$

The operator N applies canonicalisation pointwise and subtracts the trivial part:

$$\begin{aligned} N &: \mathcal{P}(\mathcal{R} \times \mathcal{V} \times \mathcal{X}) \to \mathcal{P}(\mathcal{R} \times \mathcal{V} \times \mathcal{X}) \\ N(\Xi) &= \kappa(\Xi) \setminus \mathrm{triv}(\kappa(\Xi)) \\ \kappa &: \mathcal{P}(\mathcal{R} \times \mathcal{V} \times \mathcal{X}) \to \mathcal{P}(\mathcal{R} \times \mathcal{V} \times \mathcal{X}) \\ \kappa(\Xi) &= \{\mathrm{norm}(O, L, R) \mid (O, L, R) \in \Xi\} \end{aligned} \tag{48}$$

By construction, N is idempotent and monotone, and it preserves the satisfiability of Ξ.

Lemma 126 **Normalisation Soundness** : For any choice of representative function rep and any substitution υ, υ satisfies Ξ iff υ satisfies $N(\Xi)$ (up to definitional equality of types).

Proof. ($\Rightarrow$) Assume υ satisfies Ξ.

- Canonicalisation. Replacing L by $\bar{L}$ (and R by $\bar{R}$ when $R \in \mathcal{V}$) preserves truth: by construction of classes (Definition 118), $L \approx \bar{L}$ holds (from explicit equalities/unifications and anti-symmetry cycles). Since υ satisfies those atoms, $\upsilon(L)$ and $\upsilon(\bar{L})$ are definitionally equal, so each constraint's truth value is unchanged.
- Trivial atoms. Every case flagged by $\mathrm{trivial}$ is valid under any υ: $\alpha <: \top$, $\bot <: \alpha$, and variable–variable atoms with endpoints already in the same class (ord/dep/unify, and eq when the RHS is a variable in the same class). Dropping such tautologies does not restrict solutions.

Hence υ satisfies $\kappa(\Xi) \setminus \mathrm{triv}(\kappa(\Xi)) = N(\Xi)$.

($\Leftarrow$) Conversely, each atom of Ξ either appears (up to representative renaming) in $\kappa(\Xi)$ or is in $\mathrm{triv}(\kappa(\Xi))$. If υ satisfies $N(\Xi)$, it satisfies the renamed atoms; reintroducing tautologies preserves satisfaction. Thus υ satisfies Ξ. □

Normalisation guarantees that every atom in $N(\Xi)$ mentions only class representatives, and that reflexive, duplicate, and tautological atoms are removed. It preserves the solution set while reducing the constraint set, thereby enabling more efficient later phases.

6.10.5.3. Constraint Propagation

Propagation pushes information along the variable graph to sharpen bounds. We work over representatives (from Chapter 6.10.5.1) and never invent new types: only joins/meets of already present bounds are produced.

Definition 127 **Propagation Graph** : Let Ξ be normalised (Chapter 6.10.5.2). Build a directed graph G over the representatives of variables:

$$
\begin{aligned}
\mathrm{edge}(\alpha, \beta) &:= (\bar{\alpha}, \bar{\beta}) \\
N &:= \{\bar{v} \mid v \in \mathrm{occ}(\Xi) \cap \mathcal{V}\} \\
E_{\leq} &:= \{\mathrm{edge}(\alpha, \beta) \mid (\mathtt{ord}, \alpha, \beta) \in \Xi\} \\
E_{\mathrm{dep}} &:= \{\mathrm{edge}(\alpha, \beta) \mid (\mathtt{dep}, \alpha, \beta) \in \Xi\} \\
E_{\mathrm{any}} &:= E_{\leq} \cup E_{\mathrm{dep}} \\
G_{\leq} &:= (N, E_{\leq}) \\
G_{\mathrm{dep}} &:= (N, E_{\mathrm{dep}}) \\
G_{\mathrm{any}} &:= (N, E_{\mathrm{any}})
\end{aligned}
\tag{49}
$$

No self-loops remain: unification (Chapter 6.10.5.1) plus normalisation (Chapter 6.10.5.2) remove trivial ord/dep atoms; equalities between variables have already been absorbed by unification.

Definition 128 **Local Accumulator** :

We collect the initial per-variable constraints from Ξ: LB_0 are lower bounds ($T <: v$), GB_0 are upper bounds ($v <: U$), and EQ_0 are equalities ($v \equiv T$) with a non-variable right-hand side.

$$\begin{aligned} \mathrm{LB}_0 &: N \to \mathcal{P}(H) \\ \mathrm{LB}_0(v) &\coloneqq \{T \mid (\mathtt{leq}, v, T) \in \Xi\} \\ \mathrm{GB}_0 &: N \to \mathcal{P}(H) \\ \mathrm{GB}_0(v) &\coloneqq \{U \mid (\mathtt{geq}, v, U) \in \Xi\} \\ \mathrm{EQ}_0 &: N \to \mathcal{P}(H) \\ \mathrm{EQ}_0(v) &\coloneqq \{T \mid (\mathrm{eq}, v, T) \in \Xi \wedge T \notin \mathcal{V}\} \end{aligned} \tag{50}$$

Equality atoms whose right-hand side is a variable are handled by unification and therefore are not accumulated here.

Definition 129 **Propagation Graph Reachability** : Let $\tau \in \{\leq, \mathrm{dep}, \mathrm{any}\}$ and $E_\tau \subseteq N \times N$ be the edge relation of G_τ.

$$\begin{aligned} \mathrm{In}_\tau &: N \to \mathcal{P}(N) \\ \mathrm{In}_\tau(b) &\coloneqq \{a \mid (a, b) \in E_\tau\} \\ \mathrm{Out}_\tau &: N \to \mathcal{P}(N) \\ \mathrm{Out}_\tau(a) &\coloneqq \{b \mid (a, b) \in E_\tau\} \end{aligned} \tag{51}$$

Let E_τ^* be the reflexive–transitive closure of E_τ.

$$\begin{aligned} \mathrm{Pred}_\tau &: N \to \mathcal{P}(N) \\ \mathrm{Pred}_\tau(b) &\coloneqq \{a \mid (a, b) \in E_\tau^*\} \\ \mathrm{Succ}_\tau &: N \to \mathcal{P}(N) \\ \mathrm{Succ}_\tau(a) &\coloneqq \{b \mid (a, b) \in E_\tau^*\} \end{aligned} \tag{52}$$

Definition 130 **Derived Bounds** : We aggregate each node's bounds with all bounds that reach it along the subtyping graph, using the lattice join/meet (Chapter 6.4).

$$\begin{aligned} &\text{join} : \mathcal{P}(H) \to H \\ &\text{join}(S) \coloneqq \begin{cases} \bot & \text{if } |S| = 0 \\ \bigsqcup_{T \in S} T & \text{otherwise} \end{cases} \\ &\text{meet} : \mathcal{P}(H) \to H \\ &\text{meet}(S) \coloneqq \begin{cases} \top & \text{if } |S| = 0 \\ \bigsqcap_{U \in S} U & \text{otherwise} \end{cases} \end{aligned} \tag{53}$$

$$\begin{aligned} &\text{LB} : N \to H \\ &\text{LB}(v) \coloneqq \text{join}\left(\text{LB}_0(v) \cup \bigcup_{a \in \text{Pred}^*_{\leq}(v)} \text{LB}_0(a) \right) \\ &\text{GB} : N \to H \\ &\text{GB}(v) \coloneqq \text{meet}\left(\text{GB}_0(v) \cup \bigcup_{b \in \text{Succ}^*_{\leq}(v)} \text{GB}_0(b) \right) \end{aligned} \tag{54}$$

Note. G_{dep} and G_{any} are not used here. They matter only operationally (to order computations in an implementation); the abstract definitions above are order-insensitive.

Definition 131 **Propagation Operator** P : Propagation replaces all explicit lower/upper bound atoms in Ξ by one summarised bound per node, using LB/GB from Chapter 6.10.5.3. Non-bound atoms (ordering, dependency, equality, selections) are preserved.

$$\begin{aligned} \Xi_0 &\coloneqq \Xi \\ &\quad \setminus \{(\texttt{leq}, L, R) \,|\, (\texttt{leq}, L, R) \in \Xi\} \\ &\quad \setminus \{(\text{geq}, L, R) \,|\, (\text{geq}, L, R) \in \Xi\} \\ P(\Xi) &\coloneqq \Xi_0 \\ &\quad \cup \{(\texttt{leq}, v, \text{LB}(v)) \,|\, v \in N\} \\ &\quad \cup \{(\text{geq}, v, \text{GB}(v)) \,|\, v \in N\} \end{aligned} \tag{55}$$

By construction:

- Idempotent: $P(P(\Xi)) = P(\Xi)$.
- Contractive on bound multiplicity: the multiset of `leq`/`geq` atoms is reduced to exactly $2 * |N|$ atoms.
- Monotone (w.r.t. bound profiles): write $\Xi_1 \preceq \Xi_2$ iff for all v, $\text{LB}_{\Xi_1}(v) \leq \text{LB}_{\Xi_2}(v)$ and $\text{GB}_{\Xi_1}(v) \geq \text{GB}_{\Xi_2}(v)$. Then $\Xi_1 \preceq \Xi_2$ implies $P(\Xi_1) \preceq P(\Xi_2)$.

6.10.5.4. Validation

Validation checks that propagated bounds are mutually consistent and decides a (partial) substitution for variables. It does not synthesise new types: choices are drawn from the already computed equalities or bounds.

Definition 132 **Representative Bounds** : Let Ξ be normalised (Chapter 6.10.5.2) and propagated (Chapter 6.10.5.3).

For each representative $v \in N$, define the unique lower and upper bounds, and the (possibly empty) set of rigid equalities:

$$\begin{aligned} L(v) &\coloneqq \iota T.(\mathtt{leq}, v, T) \in \Xi \\ U(v) &\coloneqq \iota W.(\mathtt{geq}, v, W) \in \Xi \\ E(v) &\coloneqq \{T \mid (\mathtt{eq}, v, T) \in \Xi\} \end{aligned} \tag{56}$$

By propagation, exactly one $(\mathtt{leq}, v, T)$ and one $(\mathtt{geq}, v, U)$ occur for each v, so $L(v)$ and $U(v)$ are well-defined. $E(v)$ may be empty.

(Here, ι is the definite description operator: $\iota x.P(x)$ denotes "the unique x such that $P(x)$" and is defined only when $\exists! x.P(x)$ holds.)

Definition 133 **Validation Conditions** : For each $v \in N$ we check:

$$\begin{aligned} \mathrm{BoundsOk}(v) &\coloneqq L(v) \leq U(v) \\ \mathrm{EqOk}(v) &\coloneqq \forall a, b \in E(v).a \equiv b \\ \mathrm{EqBoundsOk}(v) &\coloneqq \forall T \in E(v).L(v) \leq T \wedge T \leq U(v) \end{aligned} \tag{57}$$

Definition 134 **Decision Function** : Validation defines a canonical witness per admissible variable and packages these choices as a (partial) substitution υ.

Define a deterministic choice-on-nonempty-sets operator:

$$\mathrm{pick} : \mathcal{P}(H) \setminus \{\varnothing\} \to H, \mathrm{pick}(S) \in S \tag{58}$$

Define the subdomain of variables N_{wit} over N that admit a witness:

$$\begin{aligned} N_{\mathrm{wit}} \coloneqq \{v \in N \mid \mathrm{BoundsOk}(v) \wedge \mathrm{EqOk}(v) \wedge \mathrm{EqBoundsOk}(v) \\ \wedge (E(v) \neq \varnothing \vee L(v) \neq \bot \vee U(v) \neq \top)\} \end{aligned} \tag{59}$$

Choose a witness for each $v \in N_{\mathrm{wit}}$, preferring more precise terms:

$$\begin{aligned} \mathrm{wit} &: N_{\mathrm{wit}} \rightharpoonup H \\ \mathrm{wit}(v) &\coloneqq \begin{cases} \mathrm{pick}(E(v)) & \text{if } E(v) \neq \varnothing \\ L(v) & \text{if } L(v) \neq \bot \\ U(v) & \text{otherwise} \end{cases} \end{aligned} \tag{60}$$

Define the validation substitution on N_{wit} by:

$$\begin{aligned} \upsilon &: N_{\mathrm{wit}} \rightharpoonup H \\ \upsilon(v) &\coloneqq \mathrm{wit}(v). \end{aligned} \tag{61}$$

Note. Determinism of pick is immaterial for semantics since EqOk ensures all elements of $E(v)$ are definitionally equal; it is required only to make the algorithmic choice stable. For every $v \in N_{\mathrm{wit}}$, wit(v) satisfies $L(v) \leq \mathrm{wit}(v) \leq U(v)$ and, when $E(v) \neq \varnothing$, we assert $\mathrm{wit}(v) \in E(v)$.

Definition 135 **Validation Diagnostics** : Let $\mathrm{mode}([\upsilon]) \in \{\mathrm{hole}, \mathrm{bound}\}$ be the class mode (Definition 118).

We accumulate a finite set of diagnostics $\mathcal{D}$ (per $\upsilon \in N$):

- Conflicting equalities: if $\neg\, \mathrm{EqOk}(\upsilon)$.
- Equality outside bounds: if $E(\upsilon) \neq \varnothing \wedge \neg\, \mathrm{EqBoundsOk}(\upsilon)$.
- Upper unsatisfiable: if $U(\upsilon) = \bot$.
- Bound violation: if $\neg\, \mathrm{BoundsOk}(\upsilon)$.
- Unconstrained hole: if $\upsilon \notin \mathrm{dom}(\mathrm{wit}) \wedge \mathrm{mode}([\upsilon]) = \mathrm{hole}$.

Bound variables may remain abstract when $\upsilon \notin \mathrm{dom}(\mathrm{wit})$.

Definition 136 **Validation Operator** V :

$$V(\Xi) \coloneqq (\upsilon, \mathcal{D}). \tag{62}$$

The range of υ is contained in $\{E(\upsilon), L(\upsilon), U(\upsilon) \mid \upsilon \in N\}$.

Validation itself does not mutate Ξ. Implementations may additionally emit a finite set of derived constraints (e.g. from $\mathrm{col}(L(\upsilon), U(\upsilon))$ or between $L(\upsilon)$ and each $T \in E(\upsilon)$) to trigger another outer iteration when helpful.

6.10.5.5. Selection

Selection resolves pending field projections and subscripts into ordinary constraints, using the current (partial) substitution from validation. It does not synthesise new types: every emitted atom is an equality or unification with a type obtained from the intrinsic $\mathrm{proj}/\mathrm{sub}$ operators (Definition 47, Definition 86).

Definition 137 **Specialisation under Substitution** : For a (partial) substitution υ obtained through Chapter 6.10.5.4, write $T[\upsilon]$ for the (capture-avoiding) simultaneous substitution of variables in $\mathrm{dom}(\upsilon)$ by their witnesses inside T.

Definition 138 **Resolution primitives** : We define a head-opacity predicate (blocks) parameterised by the current (partial) substitution υ. It returns 1 iff resolving the outer constructor would require a still-unresolved subject variable.

$$\text{blocks} : (\upsilon : \mathcal{V} \rightharpoonup H) \times H \to \{1, 0\}$$

$$\text{blocks}(\upsilon, T) \coloneqq \begin{cases} 1 & \text{if } T[\upsilon] \in \mathcal{V} \wedge T[\upsilon] \notin \text{dom}(\upsilon) \\ \text{blocks}(\upsilon, T_1) \vee \text{blocks}(\upsilon, T_2) & \text{if } T[\upsilon] = T_1 \uplus T_2 \\ \text{blocks}(\upsilon, T_1) \vee \text{blocks}(\upsilon, T_2) & \text{if } T[\upsilon] = T_1 \cap T_2 \\ \text{blocks}(\upsilon, A) & \text{if } T[\upsilon] = o\langle s | A \rangle \\ \text{blocks}(\upsilon, B) & \text{if } T[\upsilon] = \mu\beta.B \\ \text{blocks}(\upsilon, A) \vee \text{blocks}(\upsilon, B) & \text{if } T[\upsilon] = AB \\ \text{blocks}(\upsilon, A) \vee \text{blocks}(\upsilon, B) & \text{if } T[\upsilon] = A[\gamma \mapsto B] \\ \text{blocks}(\upsilon, B) & \text{if } T[\upsilon] = \Lambda\gamma : K.B \\ \text{blocks}(\upsilon, B) & \text{if } T[\upsilon] = \forall\gamma.B \\ 0 & \text{otherwise} \end{cases} \tag{63}$$

Given υ, define partial resolvers that inspect the specialised subjects:

$$\text{resolve}_{\text{proj}}(\upsilon, T, \ell) \coloneqq \begin{cases} (\texttt{pending}, \bot) & \text{if blocks}(\upsilon, T) \\ (\text{ok}, \text{proj}(T[\upsilon], \ell)) & \text{if } (T[\upsilon], \ell) \in \text{dom(proj)} \\ (\texttt{error}, \bot) & \text{otherwise} \end{cases}$$

$$\text{resolve}_{\text{sub}}(\upsilon, T, K) \coloneqq \begin{cases} (\texttt{pending}, \bot) & \text{if blocks}(\upsilon, T) \vee \text{blocks}(\upsilon, K) \\ (\text{ok}, \text{sub}(T[\upsilon], K[\upsilon])) & \text{if } (T[\upsilon], K[\upsilon]) \in \text{dom(sub)} \\ (\texttt{error}, \bot) & \text{otherwise} \end{cases} \tag{64}$$

Definition 139 **Selection Resolution** : Let Ξ be the current constraint set and υ the validation substitution from Chapter 6.10.5.4.

Partition Ξ into selection and non-selection atoms:

$$\begin{aligned} \Xi_{\texttt{sel}} &= \{(O, L, R) \in \Xi \mid O = \texttt{sel}\} \\ \Xi_{\neg,\texttt{sel}} &= \{(O, L, R) \in \Xi \mid O \neq \texttt{sel}\} \end{aligned} \tag{65}$$

Split the selection atoms into projections and subscripts:

$$\begin{aligned} \Xi_{\text{proj}} &= \{(\alpha, T, \ell) \mid (\texttt{sel}, \alpha, (\text{proj}, T, \ell)) \in \Xi_{\texttt{sel}}\} \\ \Xi_{\text{sub}} &= \{(\alpha, T, K) \mid (\texttt{sel}, \alpha, (\text{sub}, T, K)) \in \Xi_{\texttt{sel}}\} \end{aligned} \tag{66}$$

Classify each selection as resolved, pending, or erroneous:

$$\begin{aligned} \Xi_{\text{proj}}^{\text{ok}} &= \left\{ \begin{cases} (\texttt{unify}, \alpha, N) & \text{if } N \in \mathcal{V} \\ (\texttt{eq}, \alpha, N) & \text{otherwise} \end{cases} \middle| (\alpha, T, \ell) \in \Xi_{\text{proj}} \wedge \text{resolve}_{\text{proj}}(\upsilon, T, \ell) = (\text{ok}, N) \right\} \\ \Xi_{\text{proj}}^{\texttt{pending}} &= \left\{ (\texttt{sel}, \alpha, (\text{proj}, T, \ell)) \middle| (\alpha, T, \ell) \in \Xi_{\text{proj}} \wedge \text{resolve}_{\text{proj}}(\upsilon, T, \ell) = (\texttt{pending}, \bot) \right\} \\ \Xi_{\text{proj}}^{\texttt{error}} &= \left\{ \alpha \middle| (\alpha, T, \ell) \in \Xi_{\text{proj}} \wedge \text{resolve}_{\text{proj}}(\upsilon, T, \ell) = (\texttt{error}, \bot) \right\} \end{aligned} \tag{67}$$

$$\begin{aligned} \Xi_{\text{sub}}^{\text{ok}} &= \left\{ \begin{cases} (\texttt{unify}, \alpha, N) & \text{if } N \in \mathcal{V} \\ (\texttt{eq}, \alpha, N) & \text{otherwise} \end{cases} \middle| (\alpha, T, K) \in \Xi_{\text{sub}} \wedge \text{resolve}_{\text{sub}}(\upsilon, T, K) = (\text{ok}, N) \right\} \\ \Xi_{\text{sub}}^{\texttt{pending}} &= \{(\texttt{sel}, \alpha, (\text{sub}, T, K)) \mid (\alpha, T, K) \in \Xi_{\text{sub}} \wedge \text{resolve}_{\text{sub}}(\upsilon, T, K) = (\texttt{pending}, \bot)\} \\ \Xi_{\text{sub}}^{\texttt{error}} &= \{\alpha \mid (\alpha, T, K) \in \Xi_{\text{sub}} \wedge \text{resolve}_{\text{sub}}(\upsilon, T, K) = (\texttt{error}, \bot)\} \end{aligned} \tag{68}$$

Update the constraint set by discharging resolved selections to ordinary constraints and retaining pending atoms:

$$\Xi_{\text{new}} \coloneqq \Xi_{\neg,\texttt{sel}} \cup \Xi_{\text{proj}}^{\text{ok}} \cup \Xi_{\text{sub}}^{\text{ok}} \cup \Xi_{\text{proj}}^{\texttt{pending}} \cup \Xi_{\text{sub}}^{\texttt{pending}} \tag{69}$$

Definition 140 **Selection Diagnostics** : We accumulate a finite set of diagnostics $\mathcal{D}$:

- for each $\alpha \in \Xi_{\text{proj}}^{\texttt{error}} \cup \Xi_{\text{sub}}^{\texttt{error}}$ emit an error that the projection/subscript is undefined for the specialised subject.
- for each atom in $\Xi_{\text{proj}}^{\texttt{pending}} \cup \Xi_{\text{sub}}^{\texttt{pending}}$ emit a transient diagnostic indicating that resolution is blocked (cleared on the next round).

Transient diagnostics surface solver states that are waiting on unresolved variables without making progress.

Definition 141 **Selection Operator** S :

$$S_\upsilon(\Xi) \coloneqq (\Xi_{\text{new}}, \mathcal{D}). \tag{70}$$

By construction (for fixed υ):

- Total: S_υ is defined on all finite Ξ.
- Idempotent: $S_\upsilon(\Xi) = (\Xi', D)$ implies $S_\upsilon(\Xi') = (\Xi', D')$ (no new work appears without a stronger υ).
- Progress: if any selection resolves to $\texttt{ok}$, then $|\Xi_{\texttt{sel}}|$ strictly decreases in Ξ_{new}.
- Monotone in υ: extending υ (larger domain) can only turn $\texttt{pending}$ into $\texttt{ok}$ or $\texttt{error}$; $\texttt{ok}$ remains $\texttt{ok}$.

7. Implementation

The compiler is organised into four distinct phases:

- *Frontend.* (Chapter 7.2) Parses the chosen surface syntax and emits the AST.
- *AST.* (Chapter 7.3) A write-optimised mutation-heavy intermediate representation (IR) that lowers special forms and canonicalises types; it is the single target of all frontends.
- *HIR.* (Chapter 7.4) A read-optimised IR reified from the AST for type checking, specialisation, and control-flow-aware optimisations.
- *Evaluation.* (Chapter 7.5) Consumes the HIR and drives execution, depending on the chosen backend.

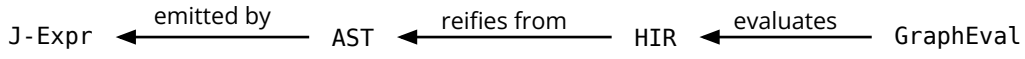


Figure 14: Compilation Pipeline

Figure 14 depicts the overall architecture and data flow between phases. Only the Frontend pushes into the next stage (Frontend → AST), because multiple surface syntaxes may exist while there is precisely one authoritative AST. From there on, each phase pulls from its single predecessor: the HIR reifies from the AST, and Evaluation consumes the HIR.

All phases follow a simple contract: take an input of the specified shape and produce an output IR with accompanying diagnostics, while remaining side-effect-free. At present, evaluation remains side-effect-free: it creates a transient graph IR (an abridged SQL query) that the graph runtime executes. Future work revisits this split so evaluation can dispatch queries directly (Chapter 8).

Each transition is a diagnostic boundary: a phase may emit diagnostics; non-fatal ones accumulate, but if a phase reports fatal diagnostics (Chapter 7.8), the pipeline halts before handing the artefact to the next phase.

7.1. Architecture

The compiler is written in Rust and organised as a set of independent crates:

- `hashql-syntax-jexpr` – J-Expr frontend.
- `hashql-ast` – AST.
- `hashql-hir` – HIR.
- `hashql-core` – cross-phase infrastructure

- `hashql-diagnostics` – diagnostic infrastructure
- `hashql-compiletest` – integration test harness

As shown in Figure 15, dependencies follow the compilation pipeline introduced in Chapter 7: the frontend depends on core and diagnostics and produces the AST; the HIR pulls from the AST; the evaluation backend consumes the HIR. Auxiliary crates (`hashql-core`, `hashql-diagnostics`, `hashql-compiletest`) span phases as needed, but the main pipeline remains intact and acyclic.

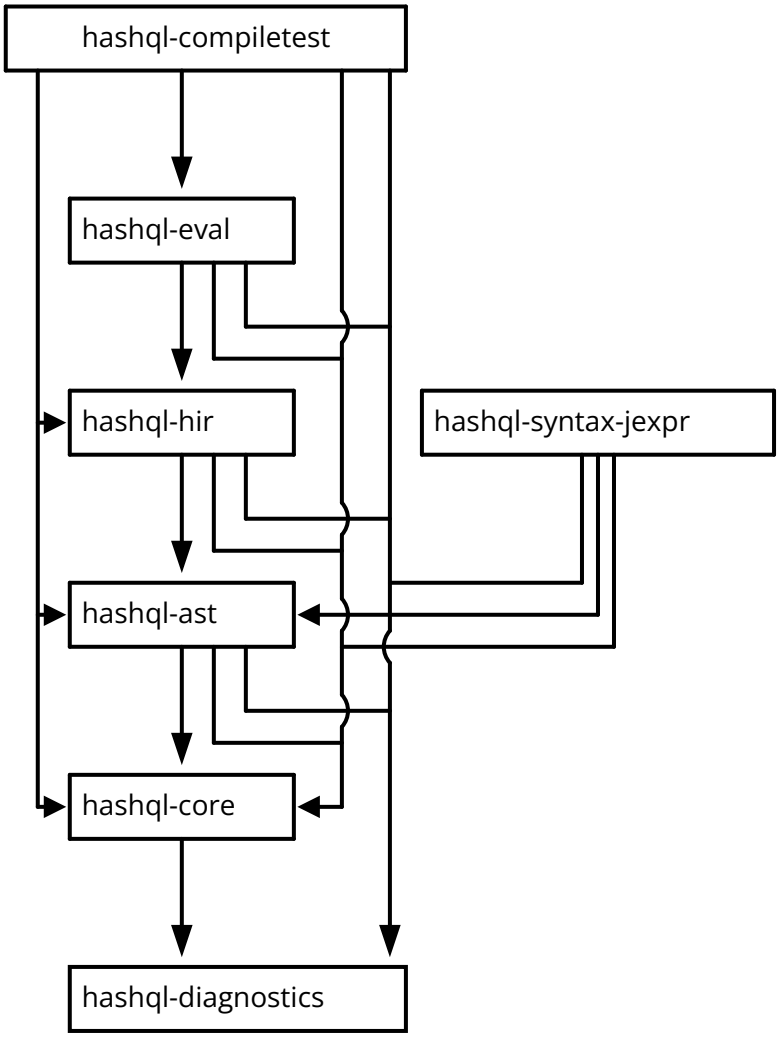


Figure 15: Crate Dependencies

7.2. Frontend

The current frontend is J-Expr (Chapter 5.1). It uses a dedicated streaming pipeline: a pull-based lexer feeds an LL(2) parser. Operating on a token stream rather than a pre-materialised JSON tree reduces allocations, enables precise location-based diagnostics, and admits JSONC (Appendix C).

Before implementing the lexer and parser, existing libraries were surveyed; none met the design goals of precise (location-based) diagnostics, incremental computation, borrowing directly from the source buffer, and JSONC support.

The lexer emits a minimal JSON token set and skips trivia (whitespace and comments): `String`, `Number`, `True`, `False`, `Null`, `Comma`, `Colon`, `LBrace`, `RBrace`, `LBracket`, `RBracket`. Comments are treated as trivia and dropped at lexing time. Trailing commas are accepted by relaxed array/object sub-parsers.

The parser is a compact state machine. Two-token lookahead is required only where J-Expr extends JSON with labelled arguments (Chapter 5.1.3): when parsing a call's argument list and encountering an object literal, the parser peeks to see if the first key is a label (a key string beginning with `:`); if so, it takes the labelled-argument production, otherwise it parses a plain object as a data expression (Chapter 5.1.4). Elsewhere, no lookahead is required; limited lookahead may be used only to enrich diagnostics, not to decide the grammar.

String literals are handled by a dedicated recursive-descent routine that parses directly over the source buffer. This choice was made for implementation simplicity; it intentionally forgoes a separate "string lexer".

The frontend pushes its result into the AST. Construction is incremental and bottom-up: nodes are assembled as soon as their children are known and written into a single allocation block, avoiding re-allocation of partial trees where possible (see arena/interning in Chapter 7.6).

7.3. Abstract Syntax Tree

The AST is a write-optimised, mutation-heavy IR that bridges frontends to the HIR. Its responsibilities are to expand special forms, canonicalise types, resolve names and imports, and assign stable node identifiers. To keep each concern small and testable, the work is split into sequential lowering passes.

On a fatal diagnostic, a pass replaces the offending subtree with `Dummy` and continues (fail-slow), so independent branches can still be analysed and reported. Downstream AST passes treat `Dummy` as inert. Immediately before reification, we assert that the tree is `Dummy`-free (by checking that no fatal diagnostics have occurred); any residual `Dummy` is an invariant violation, and the HIR aborts.

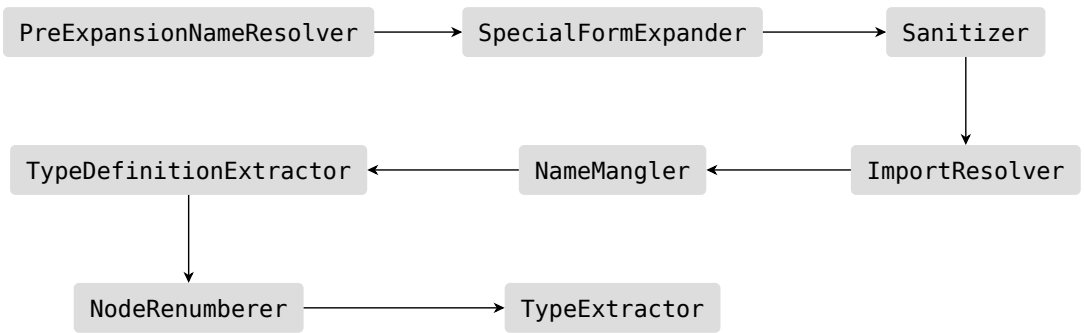


Figure 16: AST Lowering Pipeline

Figure 16 shows the passes applied to the AST.

PreExpansionNameResolver. Resolves references to special forms and rewrites them to absolute paths; these paths guide `SpecialFormExpander`. A limitation is that imports (Chapter 5.5.7) are not considered for alias tracking (import is itself a special form handled later).

SpecialFormExpander. Traverses bottom-up and rewrites calls to special forms into dedicated AST nodes according to the signatures in Chapter 5.5. After this step, remaining references to special forms are errors.

Sanitizer. Asserts the post-expansion invariants: (i) special-form identifiers no longer appear in the tree; (ii) generic constraints are not attached to path syntax. Stray special-form paths are replaced by `Dummy` with a fatal diagnostic. Stray generic constraints are stripped (turning the constraint into a plain generic) and reported as fatal diagnostics.

ImportResolver. Eliminates imports in place: within the `use` body, every referenced item is rewritten to its absolute path according to Algorithm 1; afterwards, the `use` node is replaced by its body. This preserves the functional semantics (no side effects outside the body) while removing the import node itself.

NameMangler. Renames every binding introduced by `let`, `type`, or `newtype` to `original:counter` (`:` is used as it isn't a valid identifier). Where `counter` is monotonically increasing per identifier. This ensures per-scope uniqueness, prevents accidental shadowing, and makes later analyses easier to implement as they do not need to consider scoping. The pass respects lexical scope and runs after `ImportResolver` to avoid renaming imported items. User-facing names in diagnostics are preserved via spans and dedicated de-mangling operations.

TypeDefinitionExtractor. Removes `Type` and `Newtype` nodes from the expression tree, converts their AST representation to the core type calculus, and installs the result into the type environment. The original node is replaced by its body; references remain unambiguous due to prior name mangling.

NodeRenumberer. Each AST node carries a placeholder id up to this point. This pass performs a single traversal and assigns fresh, monotonically increasing ids, producing a stable, dense id space so later phases can index side tables by node id.

TypeExtractor. Removes remaining surface-level type syntax (e.g. explicit annotations in `let`, Chapter 5.5.4) and converts it to the internal type representation used by the type system. Because this information is stored alongside the tree, it requires the stable ids produced by `NodeRenumberer`.

Once these passes have completed successfully, the HIR reifies the AST.

A future iteration (Chapter 8) may merge `PreExpansionNameResolver`, `SpecialFormExpander`, and `ImportResolver` into a single pass. The present split reflects historical order; the `SpecialFormExpander` predates the module system (Chapter 5.6), and the division kept the initial implementation small. Unifying them would reduce duplicate work and enable importing special forms directly, removing brittle alias propagation on non-specialised nodes at the cost of a more complex implementation.

7.4. High-Level Intermediate Representation

The HIR is a read-optimised IR that – unlike the AST – relies heavily on interning. It restructures the set of node types in the tree: frontend-only constructs (e.g., imports) are removed, and specialised nodes are introduced by later passes, while retaining an explicit tree to model control flow. Therefore, type checking, specialisation, linting, and control-flow-aware passes operate over the HIR. The current pipeline implements only the passes required for evaluation; additional passes are deferred to Chapter 8.

The HIR pipeline is organised into three diagnostic boundaries: pre-type-checking, type-checking, and post-type-checking. Within a boundary, passes are fail-slow; they traverse the program and accumulate diagnostics. The pipeline does not cross a boundary if any fatal diagnostic is produced inside it. To maximise diagnostic yield per run, fallible work before type checking is kept to a minimum.

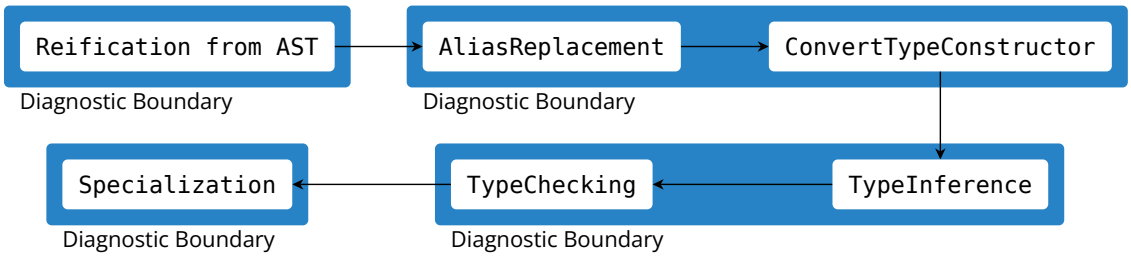


Figure 17: HIR lowering pipeline

Figure 17 shows the pipeline of HIR passes, with reification from the AST as the entry point.

`Reification.` The HIR reifies the AST by traversing the tree, converting nodes and interning them as they are created. A small set of constructs is currently unsupported (see Chapter 8): conditional expressions; struct, tuple, dict, and list literals; and labelled arguments. Each unsupported node emits a fatal diagnostic; after traversal, if any fatal diagnostic was emitted, the pipeline halts at this boundary. During reification, type annotations are desugared into dedicated HIR nodes. Listing 46 illustrates the transformation.

```JSON
["let", "foo", "Integer", "bar",
"foo"]

// after desugaring:
["let", "foo", ["as", "bar", "Integer"],
"foo"]
```

Listing 46: `let` type annotation desugaring

`AliasReplacement.` Eliminates trivial, semantics-preserving rebindings. If a binding's right-hand side is a bare path to another binding, all uses in its scope are rewritten to the target, and the let itself is replaced by its body. The pass respects lexical scope and never crosses shadowing. This removes indirection in the HIR and reduces work for later passes. Listing 47 shows a minimal example.

```JSON
["let", "foo", "bar",
"foo"
]

// after AliasReplacement:
"bar"
```

Listing 47: `AliasReplacement` example

`ConvertTypeConstructor`. Recognises paths that denote a type constructor – whether imported or locally bound – and rewrites the path to a dedicated `TypeConstructor` node. The node exists to support the current evaluator, which cannot invoke closures directly; once closures are supported (Chapter 8), this pass (and node) can be removed.

`TypeInference`. Collects constraint obligations over the HIR (Chapter 6.10.2) and invokes the solver (Chapter 6.10.5). On failure, compilation aborts; otherwise, it yields the substitution υ for use by checking.

`TypeChecking`. Traverses bottom-up using υ to substitute and simplify types, then checks each subtyping obligation. The pass is fail-slow: it visits the whole tree and accumulates violations. After traversal, if any fatal diagnostic was produced, the pipeline halts at this boundary.

`Specialization`. Rewrites recognised intrinsics to dedicated nodes (for example, arithmetic and graph operations) so the evaluator can dispatch them directly. At present, only the intrinsics implemented by the graph evaluator are specialised; references to unsupported intrinsics are reported as errors. This restriction will be lifted once the VM and additional backends are in place.

After these passes complete successfully, the chosen evaluator consumes the HIR and drives execution.

7.5. Evaluation

The current backend compiles the HIR into a transient graph IR (an abridged SQL query) executed by the graph runtime. Because it targets the existing filter engine, it inherits that engine's limitations: only a subset of valid queries can be expressed today (see Chapter 8).

The evaluator traverses the HIR bottom-up and builds the graph IR incrementally. During traversal, it maintains two value representations: (i) immediates (literals and data expressions, Chapter 5.1.4) and (ii) paths into objects, assembled via `index` (Chapter 5.5.11) and `access` (Chapter 5.5.10). Type checking in the HIR guarantees that referenced paths exist and that operand types at comparison sites are compatible.

The underlying graph IR separates statements from expressions: the top level is a boolean formula (composed with `and`/`or`/`not`), and its atoms are comparison expressions over paths and immediates. Boolean combinators cannot appear inside comparison subtrees; consequently, some semantically valid nestings cannot be lowered and are reported as compilation errors. This is a temporary limitation (see Chapter 8).

After construction, the graph IR is handed to the existing graph database infrastructure for execution. Lifting the statement/expression split and expanding supported constructs are planned in Chapter 8 and will require changes to the current query engine.

7.6. Core

The core provides cross-phase utilities and foundations used by all phases. It consists of several independent sub-modules:

`Type System`. Implements the type calculus described in Chapter 6 and is used by the AST and HIR.

`Allocator`. Most compiler data is short-lived and phase-scoped, so we use an arena-backed allocator (bump allocation in large blocks). This introduces a dedicated arena lifetime (`'heap`) on data types, but amortises allocation costs and enables reuse of allocation blocks across phases and repeated compilations. In multi-module projects, capacity tends to converge quickly, reducing allocator churn.

`Symbol`. Provides an interned `Symbol` for identifiers, keywords, and literal atoms. Interpreted through the arena, symbols support `O(1)` equality and deduplicate strings across phases.

`Interning`. Provides a generic interner for immutable values, used across the type system and the HIR.

`Span`. The implementation is syntax-agnostic, so span metadata is frontend-defined. During lexing, frontends allocate `SpanId`s from a push-only arena; subsequent phases carry only the `SpanId`. When emitting diagnostics, the reporter resolves a `SpanId` through the frontend's span table to recover source metadata. For J-Expr, that metadata includes the byte range and a JSON Pointer [14].

`Value`. Represents literal values with structural sharing. Used by the HIR evaluator to carry immediate values during query construction.

In addition, the core houses small utilities (numeric helpers, pretty-printing, etc.) that are reused across phases.

7.7. Testing Strategy

Unit tests and snapshot tests exercise the compiler. Unit tests cover isolated components (e.g., interning, the type system, etc.). Snapshot tests drive end-to-end compilation and compare emitted diagnostics and outputs to reference snapshots. Overall coverage is about 80% across crates.

To scale snapshot testing, we provide a parallelised `compiletest` harness. It traverses test directories; each directory contains a `.spec.toml` that describes the suite. For each source file, the harness either compares the run against the existing `.stdout`/`.stderr` snapshots or blesses by writing those snapshots to disk.

Tests use JSONC comments as assertions. We distinguish two kinds:

- Directives (`//@ ...`) configure the test (description, run mode: pass/fail/skip).
- Inline annotations (`//~ ...`) attach expectations to specific lines.

A case fails if an expected diagnostic is missing or if an unexpected diagnostic appears. The system and syntax are inspired by Rust's compiletest framework [48:2.4.1]. Listing 48 shows a failing case with an error annotation on Listing 48-6.

```JSON
//@ run: fail
//@ description: Ensure that special forms in invalid value positions are sanitized and error out
[
  "add",
  "::kernel::special_form::let",
  //~^ ERROR Special form cannot be used as a value
  "a"
]
```

Listing 48: Example `compiletest` file

The harness runs tests in parallel and reports a summary of failed tests as well as their missing/extraneous diagnostics.

In total, the project includes ~350 `compiletest` cases and ~1300 other tests.

7.8. Diagnostics

Diagnostics are first-class, rendering-neutral artefacts. A diagnostic is constructed once and can be rendered as HTML, terminal output, or JSON without loss of information.

A diagnostic has the following structure:

- Category. A stable, hierarchical identifier (e.g., `hir::type::mismatch`). Categories are organised by phase (J-Expr, AST, HIR, type system, evaluation). At the time of writing, 121 categories are defined.
- Severity. One of Ice, Fatal, Error, Warning, Note, or Debug. Severities carry a fatality policy used by the pipeline:
  - Fatal: Ice, Fatal, Error. These stop the pipeline at diagnostic boundaries (Chapter 7, Figure 17).
  - Non-fatal: Warning, Note, Debug. These accumulate and never block progress on their own.
- Message (optional). A short, human-readable summary.
- Labels (one or more). Each label attaches text to a span and has a role:
  - primary – the primary site of the problem (always the first label),
  - secondary – related context.
- Notes (zero or more). Explanatory text that clarifies why the issue occurred (semantics, background, cross-references).
- Help (zero or more). Actionable guidance that suggests how to fix the issue (edits, alternatives, links).

We intentionally separate notes from help: notes justify, helps instruct. Renderers present them differently (e.g., "note:" vs "help:" blocks).

8. Future Work

This work lays the foundation for the system by implementing the core pipeline and evaluator required by our initial requirements. Next, we remove current limits and align execution and expressivity with the system's goals, moving beyond the filter-based iteration's constraints.

The first step is to lift the HIR reification bans and admit conditional expressions; struct, tuple, dict, and list literals. Labelled arguments remain out of scope unless a compelling use emerges in the short term. Although the current evaluator cannot lower these constructs, enabling them in the HIR is a prerequisite for subsequent stages.

With the HIR unblocked, we will introduce a mid-level intermediate representation (MIR) in static single assignment (SSA) form, partitioned into basic blocks with control flow expressed by block edges. The surface language stays functional; SSA is an internal form that enables conventional optimisation and precise analysis. Earlier choices (e.g. name mangling) anticipated this transition.

Basic blocks also unlock a core requirement: highly parallel execution and heterogeneous dispatch. Referential-transparency and explicit control flow make dependencies explicit; a scheduler with a cost model can place blocks on available resources. This split also lets the scheduler decide, at dispatch time, which graph subqueries should be treated as local and which as global, and route them accordingly.

To execute arbitrary HashQL – not just graph queries – we will add a VM that runs code compiled from the MIR, including closures. This removes evaluator restrictions and provides a general execution path for host-side computations (e.g. data conversions) outside a graph context.

The query stack will evolve in tandem. The current `SelectCompiler` will be replaced by a richer graph low-level intermediate representation (LIR) that can express the full set of graph queries, eliminating the present statement/expression split. A second LIR will target VM execution, cleanly separating program semantics from any single database backend.

Additional possible future work (in no particular order):

- Extend the possible items that can be queried, such as entity-types and property-types, requiring code-generation from existing Rust types.
- Add type narrowing through guards (fully typed `if` constructs).
- Make indexing total by returning Option.

- Add linting passes (dead code, naming).
- Add optimisation passes (dead-code elimination, constant folding, inlining).
- Consolidate AST special forms.
- Support Currying.
- Support for labelled arguments to allow for more self-explanatory function calls.
- Add user-defined modules.
- Add user-defined macros.
- De-specialise `List`/`Dict`: express indexing via typeclass constraints.

Together, lifting HIR restrictions, adding MIR + VM, and introducing dedicated LIRs removes the constraints of the current evaluator, broadens the optimisation and allows us to express and implement the requirements initially set out.

9. Conclusion

This work lays a coherent foundation for HashQL: a language (Chapter 5) and a compiler (Chapter 7) built on precise semantics, a principled type discipline (Chapter 6) with advanced inference (Chapter 6.10), and a modular pipeline. The design emphasises separation of concerns (frontend → AST → HIR → evaluation) and a compiler that not only compiles correct programs but also assists users in resolving issues. Together, these choices favour clarity, testability, and predictable evolution.

The result is not merely theoretical: the pipeline compiles real programs end-to-end and has been integrated into the existing HASH graph; the current evaluator demonstrates feasibility and validates the design of HashQL. The surrounding infrastructure – diagnostics and an extensive test suite – supports robust evolution, while keeping the implementation deliberately conservative where the surface area is still evolving.

From this base, the path forward is clear: lift the temporary HIR restrictions, introduce a MIR in SSA form, add a VM for general execution, and evolve the graph LIR so queries no longer inherit legacy limits. These extensions build directly on the scaffolding established here and are outlined in Chapter 8.

Bibliography

[1] openCypher · openCypher. Retrieved 4 July 2025 from https://opencypher.org/

[2] Schema.org - schema.org. Retrieved 17 August 2025 from https://schema.org/

[3] 2023. Information technology - Database languages SQL - Part 1: Framework (SQL/Framework). Retrieved 4 July 2025 from https://www.iso.org/standard/76583.html

[4] 2024. Information technology - Database languages - GQL. Retrieved 4 July 2025 from https://www.iso.org/standard/76120.html

[5] Amos Wenger. 2025. facet-rs/facet. Retrieved 20 July 2025 from https://github.com/facet-rs/facet

[6] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Computing Surveys* 40, 1 (February 2008), 1–39. https://doi.org/10.1145/1322432.1322433

[7] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2018. Foundations of Modern Query Languages for Graph Databases. *ACM Computing Surveys* 50, 5 (September 2018), 1–40. https://doi.org/10.1145/3104031

[8] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (January 2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001

[9] Tim Berners-Lee, Roy T. Fielding, and Larry M. Masinter. 2005. *Uniform resource identifier (URI): generic syntax*. https://doi.org/10.17487/RFC3986

[10] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2024. Demystifying graph databases: analysis and taxonomy of data organization, system designs, and graph queries. *ACM Computing Surveys* 56, 2 (February 2024), 1–40. https://doi.org/10.1145/3604932

[11] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: design and implementation. *Journal of Functional Programming* 23, 5 (September 2013), 552–593. https://doi.org/10.1017/S095679681300018X

[12] Tim Bray. 2017. *The JavaScript object notation (JSON) data interchange format*. https://doi.org/10.17487/RFC8259

[13] N. G de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (proceedings)* 75, 5 (January 1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

[14] Paul C. Bryan, Kris Zyp, and Mark Nottingham. 2013. *JavaScript object notation (JSON) pointer*. https://doi.org/10.17487/RFC6901

[15] Jaewook Byun, Sungpil Woo, and Daeyoung Kim. 2020. ChronoGraph: Enabling Temporal Graph Traversals for Efficient Informat ion Diffusion Analysis over Time. *IEEE Transactions on Knowledge and Data Engineering* 32, 3 (March 2020), 424–437. https://doi.org/10.1109/tkde.2019.2891565

[16] Chris Bizer and Richard Cyganiak. 2014. RDF 1.1 TriG. Retrieved 15 December 2024 from https://www.w3.org/TR/2014/REC-trig-20140225/

[17] Stephen A. Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing - STOC '71*, 1971. ACM Press, Shaker Heights, Ohio, United States, 151–158. https://doi.org/10.1145/800157.805047

[18] Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*, January 25, 1982. Association for Computing Machinery, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176

[19] Dan Brickley and R.V. Guha. 2014. RDF Schema 1.1. Retrieved 18 August 2025 from https://www.w3.org/TR/rdf11-schema/

[20] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. 2014. RDF 1.1 Turtle. Retrieved 15 December 2024 from https://www.w3.org/TR/2014/REC-turtle-20140225/

[21] Martin J. Dürst and Michel Suignard. 2005. *Internationalized resource identifiers (IRIs)*. https://doi.org/10.17487/RFC3987

[22] Ecma International, TC39. 2025. *ECMAScript® 2025 language specification*. Retrieved from https://262.ecma-international.org/16.0/index.html

[23] Elixir Core Developers. 2025. Standard Library. Retrieved 27 July 2025 from https://hexdocs.pm/elixir/1.18.4

[24] Enzo. 2024. openCypher will pave the road to GQL for cypher implementers. Retrieved 4 July 2025 from https://neo4j.com/blog/cypher-and-gql/opencypher-gql-cypher-implementation/

[25] Robert Harper. 2016. *Practical foundations for programming languages* (Second edition ed.). Cambridge University Press, New York. https://doi.org/10.1017/CBO9781316576892

[26] Pavol Hell and Jaroslav Nešetřil. 1990. On the complexity of H-coloring. *Journal of Combinatorial Theory Series B* 48, 1 (February 1990), 92–110. https://doi.org/10.1016/0095-8956(90)90132-J

[27] Holger Knublauch and Dimitris Kontokostas. 2017. Shapes Constraint Language (SHACL). Retrieved 15 December 2024 from https://www.w3.org/TR/shacl/

[28] Jiamin Hou, Zhanhao Zhao, Zhouyu Wang, Wei Lu, Guodong Jin, Dong Wen, and Xiaoyong Du. 2023. AeonG: An Efficient Built-in Temporal Support in Graph Databases. *arXiv.org* (2023). https://doi.org/10.48550/ARXIV.2304.12212

[29] JeanHeyd Meneide. 2023. *A Mirror for Rust: Compile-Time Reflection Report*. Retrieved 20 July 2025 from https://soasis.org/posts/a-mirror-for-rust-a-plan-for-generic-compile-time-introspection-in-rust/

[30] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004. IEEE, San Jose, CA, USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[31] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, Pierre-Antoine Champin, and Niklas Lindström. 2020. JSON-LD 1.1. Retrieved 18 August 2025 from https://www.w3.org/TR/json-ld/

[32] Massachusetts Institute of Technology. 2023. *MIT/GNU Scheme Reference*. Retrieved 27 July 2025 from https://www.gnu.org/software/mit-scheme/documentation/stable/mit-scheme-ref/index.html

[33] Maria Massri, Zoltan Miklos, Philippe Raipin, and Pierre Meye. 2023. Clock-G: Temporal Graph Management System. *Lecture Notes in Computer Science*, 1–40. https://doi.org/10.1109/ICDE53745.2022.00215

[34] Matthew Flatt and PLT. 2010. *Reference: Racket*. Retrieved 27 July 2025 from https://racket-lang.org/tr1/

[35] Microsoft. 2025. TypeScript Handbook. Retrieved 18 August 2025 from https://www.typescriptlang.org/docs/handbook/intro.html

[36] Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* 17, 3 (December 1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

[37] Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (July 1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

[38] Oracle Coperation. 2025. java.lang.reflect (Java SE 24 API). Retrieved 20 July 2025 from https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/lang/reflect/package-summary.html

[39] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press, Cambridge, Massachusetts.

[40] Benjamin C. Pierce (Ed.). 2005. Advanced topics in types and programming languages.

[41] François Pottier. 1996. Simplifying subtyping constraints. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, June 15, 1996. Association for Computing Machinery, New York, NY, USA, 122–133. https://doi.org/10.1145/232627.232642

[42] François Pottier. 2001. Simplifying subtyping constraints: a theory. *Information and Computation* 170, 2 (November 2001), 153–183. https://doi.org/10.1006/inco.2001.2963

[43] Shriram Ramesh, Animesh Baranawal, and Yogesh Simmhan. 2021. Granite: A distributed engine for scalable path queries over temporal property graphs. *Journal of Parallel and Distributed Computing* 151, (May 2021), 94–111. https://doi.org/10.1016/j.jpdc.2021.02.004

[44] Richard Cyganiak, David Wood, and Markus Lanthaler. 2014. RDF 1.1 Concepts and Abstract Syntax. Retrieved 15 December 2024 from https://www.w3.org/TR/rdf11-concepts/

[45] Marko A. Rodriguez. 2015. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages (DBPL 2015)*, October 27, 2015. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2815072.2815073

[46] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*, June 07, 2008. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

[47] Rust Project Developers. 2025. Standard Library. Retrieved 1 August 2025 from https://doc.rust-lang.org/1.88.0/std/index.html

[48] Rust Project Developers. 2025. *Rust Compiler Development Guide*. Rust Project. Retrieved 17 August 2025 from https://rustc-dev-guide.rust-lang.org/

[49] Rust Project Developers. 2025. *The Rust Reference*. Rust Project. Retrieved 20 July 2025 from https://doc.rust-lang.org/1.88.0/reference/

[50] Alexandros Spitalas and Kostas Tsichlas. 2023. MAGMA: proposing a massive historical graph management system. *Algorithmic Aspects of Cloud Computing 13799*, 42–57. https://doi.org/10.1007/978-3-031-33437-5_3

[51] Steve Harris, Garlik and Andy Seaborne. 2013. SPARQL 1.1 Query Language. Retrieved 15 December 2024 from https://www.w3.org/TR/sparql11-query/

[52] Martin Sulzmann, Martin Odersky, and Martin Wehr. 1996. *Type inference with constrained types*. Karlsruhe. https://doi.org/10.5445/IR/26696

[53] Norbert Tausch, Michael Philippsen, and Josef Adersberger. 2011. A statically typed query language for property graphs. In *Proceedings of the 15th Symposium on International Database Engineerin g & Applications - IDEAS '11*, 2011. ACM Press, 219. https://doi.org/10.1145/2076623.2076653

[54] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN*

International Conference on Functional Programming (ICFP '14), August 19, 2014. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

[55] Dei Vilkinsons, Ciaran Morinan, Tim Diekmann, and HASH. 2025. HASH. Retrieved 17 August 2025 from https://github.com/hashintel/hash

[56] W3C OWL Working Group. 2012. OWL 2 Web Ontology Language. Retrieved 18 August 2025 from https://www.w3.org/TR/owl2-overview/

[57] Philip Wadler. 1995. Monads for functional programming. In *Advanced Functional Programming*, 1995. Springer, Berlin, Heidelberg, 24–52. https://doi.org/10.1007/3-540-59451-5_2

[58] Austin Wright, Henry Andrews, Ben Hutton, and Greg Dennis. 2022. *JSON schema: a media type for describing JSON documents*. Retrieved 18 August 2025 from https://datatracker.ietf.org/doc/draft-bhutton-json-schema-01

[59] Wyatt Childers, Peter Dimov, Barry Revzin, Andrew Sutton, Faisal Vali, and Daveed Vandevoorde. 2024. *Reflection for C++26*. Retrieved 20 July 2025 from https://isocpp.org/files/papers/P2996R4.html

[60] Hongwei Xi and Frank Pfenning. 1999. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*, January 01, 1999. Association for Computing Machinery, New York, NY, USA, 214–227. https://doi.org/10.1145/292540.292560

[61] XTDB Developers. 2025. XTQL Queries. Retrieved 2 August 2025 from https://docs.xtdb.com/reference/main/xtql/queries.html

[62] Hong Yul Yang, Ewan Tempero, and Hayden Melton. 2008. An empirical study into use of dependency injection in java. In *19th Australian Conference on Software Engineering (aswec 2008)*, March 2008. 239–247. https://doi.org/10.1109/ASWEC.2008.4483212

[63] Fu Zhang, Zhiyin Li, Dunhong Peng, and Jingwei Cheng. 2021. RDF for temporal data management – a survey. *Earth Science Informatics* 14, 2 (June 2021), 563–599. https://doi.org/10.1007/s12145-021-00574-w

[64] Zig Project Developers. 2025. *Zig Language Reference*. Zig Project. Retrieved 20 July 2025 from https://ziglang.org/documentation/0.14.1/

A. GQL Example Query

Query: Who rented a home at “Musterstraße 12” from “Musterwohnen AG”?

```
MATCH (tenant:Entity) <-[:START]- (tenancy:Entity) -[:END]->
(landlord:Entity)
WHERE
  EXISTS {
    MATCH (tenancy) -[:TYPE]-> (tenancyType:EntityType)
    MATCH (tenancyType) -[:INHERITS_FROM*]-> (linkType:EntityType)
    WHERE
      tenancyType."$id" = "https://example.com/@john/types/entity-type/
      tenancy/v/1"
      AND linkType."$id" = "https://blockprotocol.org/@blockprotocol/types/
      entity-type/link/v/1"
      // temporal pinning
      AND tenancyType.temporal.decisionTime.start <= "2024-12-12T00:00:00Z"
      AND (tenancyType.temporal.decisionTime.start +
      tenancyType.temporal.decisionTime.duration) >= "2024-12-12T00:00:00Z"
      AND linkType.temporal.decisionTime.start <= "2024-12-12T00:00:00Z"
      AND (linkType.temporal.decisionTime.start +
      linkType.temporal.decisionTime.duration) >= "2024-12-12T00:00:00Z"
  } AND EXISTS {
    MATCH (tenant) -[:TYPE]-> (tenantType:EntityType)
    WHERE
      tenantType."$id" = "https://example.com/@john/types/entity-type/tenant/
      v/1"
      // temporal pinning
      AND tenantType.temporal.decisionTime.start <= "2024-12-12T00:00:00Z"
      AND (tenantType.temporal.decisionTime.start +
      tenantType.temporal.decisionTime.duration) >= "2024-12-12T00:00:00Z"
  } AND EXISTS {
    MATCH (landlord) -[:TYPE]-> (landlordType:Entitype)
    WHERE
      landlordType."$id" = "https://example.com/@john/types/entity-type/
      landlord/v/1"
      // temporal pinning
      AND landlordType.temporal.decisionTime.start <= "2024-12-12T00:00:00Z"
```

GraphQL

```
      AND (landlordType.temporal.decisionTime.start +
      landlordType.temporal.decisionTime.duration) >= "2024-12-12T00:00:00Z"
} AND EXISTS {
  MATCH (tenancy) <-[:START]- (occupancy:Entity) -[:END]->
  (property:Entity)
  WHERE
    EXISTS {
      MATCH (occupancy) -[:TYPE]-> (occupancyType:EntityType)
      MATCH (occupancyType) -[:INHERITS_FROM*]-> (linkType:EntityType)
      WHERE
        occupancyType."$id" = "https://example.com/@john/types/entity-type/
        occupancy/v/1"
        AND linkType."$id" = "https://blockprotocol.org/@blockprotocol/
        types/entity-type/link/v/1"
        // temporal pinning
        AND occupancyType.temporal.decisionTime.start <=
        "2024-12-12T00:00:00Z"
        AND (occupancyType.temporal.decisionTime.start +
        occupancyType.temporal.decisionTime.duration) >=
        "2024-12-12T00:00:00Z"
        AND linkType.temporal.decisionTime.start <= "2024-12-12T00:00:00Z"
        AND (linkType.temporal.decisionTime.start +
        linkType.temporal.decisionTime.duration) >= "2024-12-12T00:00:00Z"
    } AND EXISTS {
      MATCH (property) -[:TYPE]-> (propertyType:EntityType)
      WHERE
        propertyType."$id" = "https://example.com/@john/types/entity-type/
        property/v/1"
        // temporal pinning
        AND propertyType.temporal.decisionTime.start <=
        "2024-12-12T00:00:00Z"
        AND (propertyType.temporal.decisionTime.start +
        propertyType.temporal.decisionTime.duration) >=
        "2024-12-12T00:00:00Z"
    }
    AND property.properties."https://example.com/@john/types/property-type/
    street/v/1" = "Musterstraße 12"
    // temporal pinning
    AND occupancy.temporal.decisionTime.start <= "2024-12-12T00:00:00Z"
    AND (occupancy.temporal.decisionTime.start +
    occupancy.temporal.decisionTime.duration) >= "2024-12-12T00:00:00Z"
    AND property.temporal.decisionTime.start <= "2024-12-12T00:00:00Z"
    AND (property.temporal.decisionTime.start +
    property.temporal.decisionTime.duration) >= "2024-12-12T00:00:00Z"
}
// temporal pinning
AND landlord.properties."https://example.com/@john/types/property-type/
name/v/1" = "Musterwohnen AG"
```

```
59    AND tenant.temporal.decisionTime.start <= "2024-12-12T00:00:00Z"
60    AND (tenant.temporal.decisionTime.start +
      tenant.temporal.decisionTime.duration) >= "2024-12-12T00:00:00Z"
61    AND tenancy.temporal.decisionTime.start <= "2024-12-12T00:00:00Z"
62    AND (tenancy.temporal.decisionTime.start +
      tenancy.temporal.decisionTime.duration) >= "2024-12-12T00:00:00Z"
63    AND landlord.temporal.decisionTime.start <= "2024-12-12T00:00:00Z"
64    AND (landlord.temporal.decisionTime.start +
      landlord.temporal.decisionTime.duration) >= "2024-12-12T00:00:00Z"
```

B. Grammar

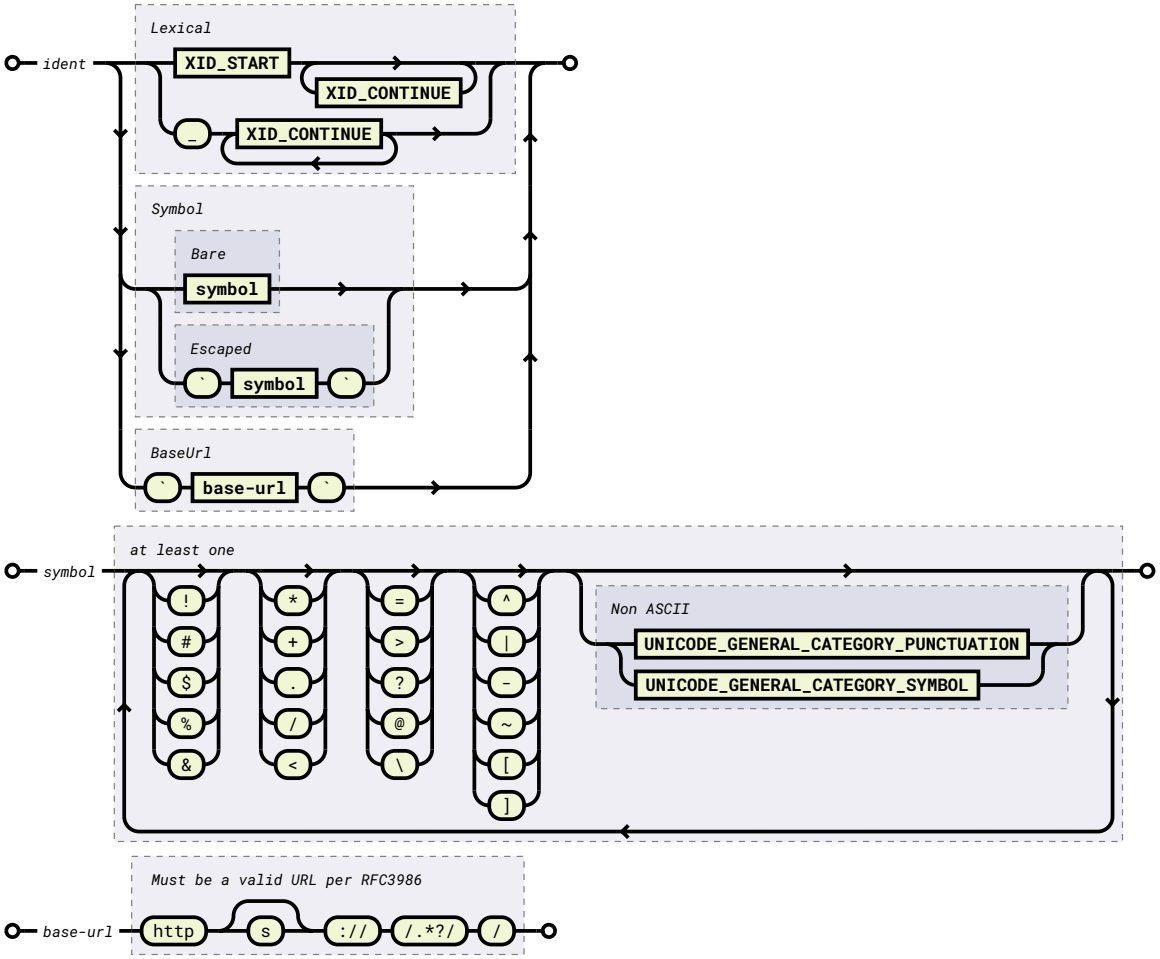
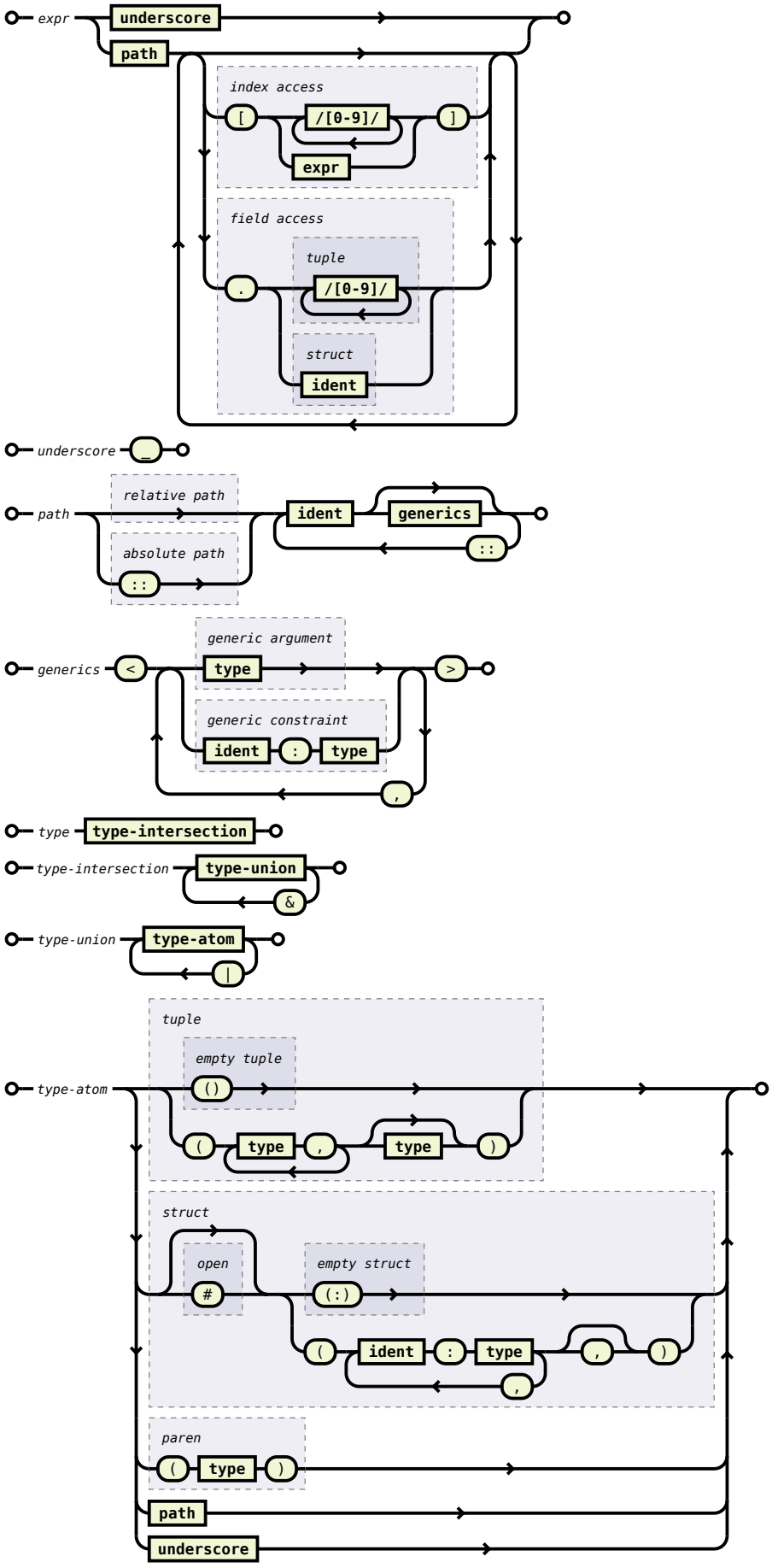


Figure 18: Identifier Grammar



Listing 49: J-Expr Embedded DSL Grammar

C. JSONC

No official standard exists for JSON with comments (JSONC); the variant described here mirrors features found in existing parsers. JSONC is treated as a strict superset of JSON, introducing two extensions: comments and optional trailing commas.

Comments follow the syntax defined in ECMAScript [22:12.5]. A single-line comment starts with `//` and continues up, but not including, to the next line terminator. A multi-line comment starts with `/*` and ends with `*/`; any characters are permitted inside, except the sequence `*/`, so multi-line comments cannot be nested.

Standard JSON forbids a comma after the final element of an array or object [12]. Our implementation of JSONC relaxes this rule by allowing exactly one trailing comma; if present, the parser accepts the input but issues a warning. Multiple trailing commas remain invalid.

D. Type System Rules

D.1. Subtyping

$$\frac{\mathsf{S} <: \mathsf{T}}{\mathsf{S} <: (T \cup U)}\ \text{S-UNION3}$$

$$\frac{\mathsf{S} <: \mathsf{U}}{\mathsf{S} <: (T \cup U)}\ \text{S-UNION2}$$

$$\frac{\mathsf{T} <: \mathsf{S} \quad \mathsf{U} <: \mathsf{S}}{(T \cup U) <: \mathsf{S}}\ \text{S-UNION1}$$

$$\frac{S_1 \coloneqq T_0 \times \ldots \times T_{n-1} \quad S_2 \coloneqq U_0 \times \ldots \times U_{n-1} \quad \mathrm{dom}(S_1) = \mathrm{dom}(S_2) \quad \forall\, \mathsf{i} \in \mathrm{dom}(S_1)\colon T_i <: U_i}{S_1 <: S_2}\ \text{S-TUP}$$

$$\frac{\mathsf{T} <: \mathsf{U} \quad \mathsf{U} <: \mathsf{V}}{\mathsf{T} <: \mathsf{V}}\ \text{S-TRANS}$$

$$\frac{}{\mathsf{T} <: \top}\ \text{S-TOP}$$

$$\frac{S_1 \coloneqq (\omega_1 \mid \ell_1 : T_1, \ldots, \ell_n : T_n) \quad S_2 \coloneqq (+1 \mid m_1 : U_1, \ldots, m_n : U_n) \quad \mathrm{dom}(S_1) \supseteq \mathrm{dom}(S_2) \quad \forall\, \ell \in \mathrm{dom}(S_2).\ \mathrm{proj}(S_1, \ell) <: \mathrm{proj}(S_2, \ell)}{S_1 <: S_2}\ \text{S-STRC2}$$

$$\frac{S_1 \coloneqq (0 \mid \ell_1 : T_1, \ldots, \ell_n : T_n) \quad S_2 \coloneqq (0 \mid m_1 : U_1, \ldots, m_n : U_n) \quad \mathrm{dom}(S_1) = \mathrm{dom}(S_2) \quad \forall\, \ell \in \mathrm{dom}(S_1).\ \mathrm{proj}(S_1, \ell) <: \mathrm{proj}(S_2, \ell)}{S_1 <: S_2}\ \text{S-STRC1}$$

$$\frac{}{\mathsf{T} <: \mathsf{T}}\ \text{S-REFL}$$

$$\frac{\langle S_1, S_2 \rangle \in \Theta}{\Delta;\, \Theta \mid\!- S_1 <: S_2}\ \text{S-REC2}$$

$$\frac{S_1 \coloneqq \mu\, \mathsf{X}.\ T_1 \quad S_2 \coloneqq \mu\, \mathsf{X}.\ T_2 \quad \langle S_1, S_2 \rangle \notin \Theta \quad \Delta, \mathsf{X}\colon *;\, \Theta, \langle S_1, S_2 \rangle \mid\!- T_1 <: T_2}{\Delta;\, \Theta \mid\!- S_1 <: S_2}\ \text{S-REC1}$$

$$\frac{s_1 = s_2 \quad \mathsf{T} <: \mathsf{U}}{o\langle s_1 | T \rangle <: o\langle s_2 | U \rangle}\ \text{S-NOM}$$

$$\frac{\mathsf{T} <: U_1 \qquad \mathsf{T} <: U_2}{\mathsf{T} <: U_1 \sqcap U_2}\ \text{S-MEET3}$$

$$\frac{}{T_1 \sqcap T_2 <: T_2}\ \text{S-MEET2}$$

$$\frac{}{T_1 \sqcap T_2 <: T_1}\ \text{S-MEET1}$$

$$\frac{T_1 <: T_2}{\mathsf{List}\ T_1 <: \mathsf{List}\ T_2}\ \text{S-LIST}$$

$$\frac{U_1 <: \mathsf{T} \qquad U_2 <: \mathsf{T}}{U_1 \sqcup U_2 <: \mathsf{T}}\ \text{S-JOIN3}$$

$$\frac{}{T_2 <: T_1 \sqcup T_2}\ \text{S-JOIN2}$$

$$\frac{}{T_1 <: T_1 \sqcup T_2}\ \text{S-JOIN1}$$

$$\frac{\mathsf{S} <: \mathsf{T} \qquad \mathsf{S} <: \mathsf{U}}{\mathsf{S} <: (T \sqcap U)}\ \text{S-INTER3}$$

$$\frac{}{(T \sqcap U) <: \mathsf{U}}\ \text{S-INTER2}$$

$$\frac{}{(T \sqcap U) <: \mathsf{T}}\ \text{S-INTER1}$$

$$\frac{\dfrac{}{\Delta \mid\!- \mathit{Integer} : *}\ \text{K-INT} \qquad \dfrac{}{\Delta \mid\!- \mathit{Number} : *}\ \text{K-NUM}}{\mathit{Integer} <: \mathit{Number}}\ \text{S-INT}$$

$$\frac{K_1 == K_2 \qquad V_1 <: V_2}{\mathsf{Dict}\ K_1\ V_1 <: \mathsf{Dict}\ K_2\ V_2}\ \text{S-DICT}$$

$$\frac{S_1 \coloneqq A_1 \to B_1 \qquad S_2 \coloneqq A_2 \to B_2 \qquad A_2 <: A_1 \qquad B_1 <: B_2}{S_1 <: S_2}\ \text{S-CLO}$$

$$\frac{}{\bot <: \mathsf{T}}\ \text{S-BOT}$$

$$\frac{\mathsf{T} <: \mathsf{U} \qquad \mathsf{U} <: \mathsf{T}}{\mathsf{T} == \mathsf{U}}\ \text{S-ANTI}$$

D.2. Variance

$$\frac{(T : v) \notin \Phi}{\Phi \mid\!- \mathsf{T}: +1}\ \text{V-VAR2}$$

$$\frac{(T : v) \in \Phi}{\Phi \mid\!- \mathsf{T}: \mathsf{v}}\ \text{V-VAR1}$$

D.3. Join

$$\frac{}{(S \uplus T) \sqcup \mathsf{U} = (S \sqcup U) \uplus (T \sqcup U)}\ \text{J-UNION}$$

$$\frac{S_1 \coloneqq T_0 \times \ldots \times T_{n-1} \quad S_2 \coloneqq U_0 \times \ldots \times U_{n-1} \quad \mathrm{dom}(S_1) = \mathrm{dom}(S_2)}{S_1 \sqcup S_2 = (T_0 \sqcup U_0 \times \ldots \times T_{n-1} \sqcup U_{n-1})}\ \text{J-TUP}$$

$$\frac{\mathsf{T} <: \mathsf{U} \quad \frac{}{\mathsf{T} <: \mathsf{T}}\ \text{S-REFL} \quad \frac{}{T_1 <: T_1 \sqcup T_2}\ \text{S-JOIN1} \quad \frac{U_1 <: \mathsf{T} \quad U_2 <: \mathsf{T}}{U_1 \sqcup U_2 <: \mathsf{T}}\ \text{S-JOIN3} \quad \frac{\mathsf{T} <: \mathsf{U} \quad \mathsf{U} <: \mathsf{T}}{\mathsf{T} == \mathsf{U}}\ \text{S-ANTI}}{\mathsf{T} \sqcup \mathsf{U} == \mathsf{U}}\ \text{J-SUB}$$

$$\frac{S_1 \coloneqq (\omega_1 \mid \ell_1 : T_1, \ldots, \ell_n : T_n) \quad S_2 \coloneqq (\omega_2 \mid m_1 : U_1, \ldots, m_n : U_n) \quad \mathrm{dom}(S_1) = \mathrm{dom}(S_2)}{S_1 \sqcup S_2 = (\omega_1 \sqcap \omega_2 \mid \ell_1 : T_1 \sqcup U_1, \ldots, \ell_n : T_n \sqcup U_n)}\ \text{J-STRC}$$

$$\frac{s_1 = s_2}{o\langle s_1 | T\rangle \sqcup o\langle s_2 | U\rangle = o\langle s_1 | T \sqcup U\rangle}\ \text{J-NOM}$$

$$\frac{T_1 <: U_1 \quad T_2 <: U_2}{T_1 \sqcup T_2 <: U_1 \sqcup U_2}\ \text{J-MONO}$$

$$\frac{}{\mathsf{List}\ T_1 \sqcup \mathsf{List}\ T_2 = \mathsf{List}\ (T_1 \sqcup T_2)}\ \text{J-LIST}$$

$$\frac{}{(S \sqcap T) \sqcup \mathsf{U} = (S \sqcup U) \sqcap (T \sqcup U)}\ \text{J-INTER}$$

$$\frac{}{\mathsf{T} \sqcup \mathsf{T} = \mathsf{T}}\ \text{J-IDEM}$$

$$\frac{\nexists\, \mathsf{S}.\ \mathsf{T} \sqcup \mathsf{U} \Downarrow \mathsf{S}}{\mathsf{T} \sqcup \mathsf{U} = \mathsf{T} \uplus \mathsf{U}}\ \text{J-FALL}$$

$$\frac{}{\mathsf{T} \sqcup (U \sqcap V) = (T \sqcup U) \sqcap (T \sqcup V)}\ \text{J-DIST}$$

$$\frac{K_1 == K_2}{\mathsf{Dict}\ K_1\ V_1 \sqcup \mathsf{Dict}\ K_2\ V_2 = \mathsf{Dict}\ K_1\ (V_1 \sqcup V_2)}\ \text{J-DICT}$$

$$\frac{}{\mathsf{T} \sqcup \mathsf{U} = \mathsf{U} \sqcup \mathsf{T}}\ \text{J-COMM}$$

$$\frac{S_1 \coloneqq A_1 \longrightarrow B_1 \quad S_2 \coloneqq A_2 \longrightarrow B_2 \quad \mathrm{ari}(S_1) = \mathrm{ari}(S_2)}{S_1 \sqcup S_2 = (A_1 \sqcap A_2) \longrightarrow (B_1 \sqcup B_2)}\ \text{J-CLO}$$

$$\frac{}{\mathsf{T} \sqcup (U \sqcup V) = (T \sqcup U) \sqcup \mathsf{V}}\ \text{J-ASSOC}$$

$$\frac{}{\mathsf{T} \sqcup (T \sqcap U) = \mathsf{T}}\ \text{J-ABSORB}$$

D.4. Meet

$$\frac{}{(S \uplus T) \sqcap \mathsf{U} = (S \sqcap U) \uplus (T \sqcap U)}\ \text{M-UNION}$$

$$\frac{S_1 \coloneqq T_0 \times \ldots \times T_{n-1} \quad S_2 \coloneqq U_0 \times \ldots \times U_{n-1} \quad \mathrm{dom}(S_1) = \mathrm{dom}(S_2)}{S_1 \sqcap S_2 = (T_0 \sqcap U_0 \times \ldots \times T_{n-1} \sqcap U_{n-1})}\ \text{M-TUP}$$

$$\frac{\mathsf{T} <: \mathsf{U} \quad \dfrac{}{\mathsf{T} <: \mathsf{T}}\,\text{S-REFL} \quad \dfrac{}{T_1 \sqcap T_2 <: T_1}\,\text{S-MEET1} \quad \dfrac{\mathsf{T} <: U_1 \quad \mathsf{T} <: U_2}{\mathsf{T} <: U_1 \sqcap U_2}\,\text{S-MEET3} \quad \dfrac{\mathsf{T} <: \mathsf{U} \quad \mathsf{U} <: \mathsf{T}}{\mathsf{T} == \mathsf{U}}\,\text{S-ANTI}}{\mathsf{T} \sqcap \mathsf{U} == \mathsf{T}}\,\text{M-SUB}$$

$$\frac{\begin{array}{c} S_1 \coloneqq \{0 \mid \ell_1 : T_1, ..., \ell_n : T_n\} \quad S_2 \coloneqq \{+1 \mid m_1 : U_1, ..., m_n : U_n\} \quad \mathrm{dom}(S_2) \subseteq \mathrm{dom}(S_1) \\ \mathsf{F} \coloneqq \{\langle l, \mathrm{proj}(S_1, \ell) \sqcap \mathrm{proj}(S_2, \ell)\rangle \mid \ell \in \mathrm{dom}(S_1) \cap \mathrm{dom}(S_2)\} \\ \mathsf{G} \coloneqq \{\langle l, \mathrm{proj}(S_1, \ell)\rangle \mid \ell \in \mathrm{dom}(S_1) \setminus \mathrm{dom}(S_2)\} \end{array}}{S_1 \sqcap S_2 = \{0 \mid F \cup G\}}\,\text{M-STRC3}$$

$$\frac{S_1 \coloneqq \{0 \mid \ell_1 : T_1, ..., \ell_n : T_n\} \quad S_2 \coloneqq \{0 \mid m_1 : U_1, ..., m_n : U_n\} \quad \mathrm{dom}(S_1) = \mathrm{dom}(S_2)}{S_1 \sqcap S_2 = \{0 \mid \langle \ell, \mathrm{proj}(S_1, \ell) \sqcap \mathrm{proj}(S_2, \ell)\rangle \mid \ell \in \mathrm{dom}(S_1) \cap \mathrm{dom}(S_2)\}}\,\text{M-STRC2}$$

$$\frac{\begin{array}{c} S_1 \coloneqq \{+1 \mid \ell_1 : T_1, ..., \ell_n : T_n\} \quad S_2 \coloneqq \{+1 \mid m_1 : U_1, ..., m_n : U_n\} \\ \mathsf{F} \coloneqq \{\langle \ell, \mathrm{proj}(S_1, \ell) \sqcap \mathrm{proj}(S_2, \ell)\rangle \mid \ell \in \mathrm{dom}(S_1) \cap \mathrm{dom}(S_2)\} \\ \mathsf{G} \coloneqq \{\langle \ell, \mathrm{proj}(S_1, \ell)\rangle \mid \ell \in \mathrm{dom}(S_1) \setminus \mathrm{dom}(S_2)\} \\ \mathsf{H} \coloneqq \{\langle \ell, \mathrm{proj}(S_2, \ell)\rangle \mid \ell \in \mathrm{dom}(S_2) \setminus \mathrm{dom}(S_1)\} \end{array}}{S_1 \sqcap S_2 = \{+1 \mid F \cup G \cup H\}}\,\text{M-STRC1}$$

$$\frac{s_1 = s_2}{o\langle s_1 | T\rangle \sqcap o\langle s_2 | U\rangle = o\langle s_1 | T \sqcap U\rangle}\,\text{M-NOM}$$

$$\frac{T_1 <: U_1 \quad T_2 <: U_2}{T_1 \sqcap T_2 <: U_1 \sqcap U_2}\,\text{M-MONO}$$

$$\frac{}{\mathsf{List}\ T_1 \sqcap \mathsf{List}\ T_2 = \mathsf{List}\ (T_1 \sqcap T_2)}\,\text{M-LIST}$$

$$\frac{}{(S \curlywedge T) \sqcap \mathsf{U} = (S \sqcap U) \curlywedge (T \sqcap U)}\,\text{M-INTER}$$

$$\frac{}{\mathsf{T} \sqcap \mathsf{T} = \mathsf{T}}\,\text{M-IDEM}$$

$$\frac{\nexists\, \mathsf{S}.\ \mathsf{T} \sqcap \mathsf{U} \Downarrow_0 \mathsf{S}}{\mathsf{T} \sqcap \mathsf{U} = \mathsf{T} \curlywedge \mathsf{U}}\,\text{M-FALL}$$

$$\frac{}{\mathsf{T} \sqcap (U \sqcup V) = (T \sqcap U) \sqcup (T \sqcap V)}\,\text{M-DIST}$$

$$\frac{K_1 \equiv K_2}{\mathsf{Dict}\ K_1\ V_1 \sqcap \mathsf{Dict}\ K_2\ V_2 = \mathsf{Dict}\ K_1\ (V_1 \sqcap V_2)}\,\text{M-DICT}$$

$$\frac{}{\mathsf{T} \sqcap \mathsf{U} = \mathsf{U} \sqcap \mathsf{T}}\,\text{M-COMM}$$

$$\frac{S_1 \coloneqq A_1 \to B_1 \quad S_2 \coloneqq A_2 \to B_2 \quad \mathrm{ari}(S_1) = \mathrm{ari}(S_2)}{S_1 \sqcap S_2 = (A_1 \sqcup A_2) \to (B_1 \sqcap B_2)}\,\text{M-CLO}$$

$$\frac{}{\mathsf{T} \sqcap (U \sqcap V) = (T \sqcap U) \sqcap \mathsf{V}}\,\text{M-ASSOC}$$

$$\frac{}{\mathsf{T} \sqcap (T \sqcup U) = \mathsf{T}}\,\text{M-ABSORB}$$

D.5. Kind

$$\frac{(T : K) \in \Delta}{\Delta \mid\!- \text{T: K}}\ \text{K-VAR}$$

$$\frac{}{\Delta \mid\!- \vec{\cup}: * \Rightarrow * \Rightarrow *}\ \text{K-UNION}$$

$$\frac{}{\Delta \mid\!- T_1 \times ... \times T_n : *}\ \text{K-TUP}$$

$$\frac{}{\Delta \mid\!- \top : *}\ \text{K-TOP}$$

$$\frac{\Delta, \alpha: K_1 \mid\!- \text{T: } K_2 \qquad \Delta \mid\!- \text{U: } K_1}{\Delta \mid\!- \text{T}[\alpha \mapsto U] : K_2}\ \text{K-SUBST}$$

$$\frac{\Delta, T_1 : K_1 \mid\!- T_2 <: T_3}{(\Lambda T_1 : K_1 . T_2) <: (\Lambda T_1 : K_1 . T_3)}\ \text{K-SUB}$$

$$\frac{}{\Delta \mid\!- (\omega \mid l_1 : T_1, ..., l_n : T_n) : *}\ \text{K-STRC}$$

$$\frac{}{\Delta \mid\!- String : *}\ \text{K-STR}$$

$$\frac{\Delta, \text{T: K} \mid\!- \text{S: K}}{\Delta \mid\!- (\Lambda \alpha : K . T)\ \text{S} == \text{T}[\alpha \mapsto S]}\ \text{K-REDUC}$$

$$\frac{\Delta, \text{X: } * \mid\!- \text{T: } * \qquad (X \text{ occurs contractively in } T)}{\Delta \mid\!- \mu\ \text{X. T: } *}\ \text{K-REC}$$

$$\frac{}{\Delta \mid\!- Number : *}\ \text{K-NUM}$$

$$\frac{}{\Delta \mid\!- Null : *}\ \text{K-NULL}$$

$$\frac{\Delta \mid\!- \text{T: } * \qquad \Delta \mid\!- \text{U: } *}{\Delta \mid\!- \text{T} \sqcap \text{U: } *}\ \text{K-MEET}$$

$$\frac{}{\Delta \mid\!- \text{List: } * \Rightarrow *}\ \text{K-LIST}$$

$$\frac{\Delta \mid\!- \text{T: } * \qquad \Delta \mid\!- \text{U: } *}{\Delta \mid\!- \text{T} \sqcup \text{U: } *}\ \text{K-JOIN}$$

$$\frac{}{\Delta \mid\!- \vec{\cap}: * \Rightarrow * \Rightarrow *}\ \text{K-INTER}$$

$$\frac{}{\Delta \mid\!- Integer : *}\ \text{K-INT}$$

$$\frac{}{\Delta \mid\!- \text{Dict: } * \Rightarrow * \Rightarrow *}\ \text{K-DICT}$$

$$\frac{\Delta \mid\!- \text{T: K} \qquad \beta \notin \text{FV}(T)}{\Delta \mid\!- \Lambda\ \alpha\text{: K . T} == \Lambda\ \beta\text{: K . T}[\alpha \mapsto \beta]}\ \text{K-CONV}$$

$$\frac{\Delta \mid\!- \text{A: } * \qquad \Delta \mid\!- \text{B: } *}{\Delta \mid\!- \text{A} \longrightarrow \text{B: } *}\ \text{K-CLO}$$

$$\frac{}{\Delta \mid\!- \bot : *}\ \text{K-BOT}$$

$$\frac{}{\Delta \mid\!- \mathit{Boolean} : *}\ \text{K-BOOL}$$

$$\frac{\Delta \mid\!- T_1 : K_1 \Rightarrow K_2 \quad \Delta \mid\!- T_2 : K_1}{\Delta \mid\!- T_1\ T_2 : K_2}\ \text{K-APP}$$

$$\frac{\Delta, T_1 : K_1 \mid\!- T_2 : K_2}{\Delta \mid\!- \Lambda\, T_1 : K_1.\ T_2 : K_1 \Rightarrow K_2}\ \text{K-ABS}$$

D.6. Term

$$\frac{(e : T) \in \Gamma}{\Gamma \mid\!- \mathsf{e} : \mathsf{T}}\ \text{T-VAR}$$

$$\frac{\Gamma \mid\!- \mathsf{e} : \mathsf{T} \qquad \dfrac{\dfrac{}{\mathsf{T} <: \mathsf{T}}\ \text{S-REFL} \qquad \dfrac{\mathsf{S} <: \mathsf{T}}{\mathsf{S} <: (T \uplus U)}\ \text{S-UNION3}}{\mathsf{T} <: (T \uplus U)} \qquad \dfrac{\Gamma \mid\!- \mathsf{e} : \mathsf{T} \quad \mathsf{T} <: \mathsf{U}}{\Gamma \mid\!- \mathsf{e} : \mathsf{U}}\ \text{T-SUB}}{\Gamma \mid\!- \mathsf{e} : \mathsf{T} \uplus \mathsf{U}}\ \text{T-UNION}$$

$$\frac{\Gamma \mid\!- e_0 : T_0 \quad \ldots \quad \Gamma \mid\!- e_{n-1} : T_{n-1}}{\Gamma \mid\!- (\!| e_0, \ldots, e_{n-1} |\!) : T_0 \times \ldots \times T_{n-1}}\ \text{T-TUP}$$

$$\frac{\dfrac{}{\Delta \mid\!- \mathit{Boolean} : *}\ \text{K-BOOL}}{\Gamma \mid\!- \mathsf{true} : \mathit{Boolean}}\ \text{T-TRUE}$$

$$\frac{\Gamma \mid\!- \mathsf{a} : \mathsf{T} \quad \Gamma \mid\!- \mathsf{k} : \mathsf{K} \quad (T, K) \in \mathrm{dom}(\mathrm{sub})}{\Gamma \mid\!- \mathsf{a}[k] : \mathrm{sub}(T, K)}\ \text{T-SUBS}$$

$$\frac{\Gamma \mid\!- \mathsf{e} : \mathsf{T} \quad \mathsf{T} <: \mathsf{U}}{\Gamma \mid\!- \mathsf{e} : \mathsf{U}}\ \text{T-SUB}$$

$$\frac{\dfrac{}{\Delta \mid\!- \mathit{String} : *}\ \text{K-STR} \qquad \mathsf{s}\text{ is a string literal}}{\Gamma \mid\!- \mathsf{s} : \mathit{String}}\ \text{T-STRLIT}$$

$$\frac{\Gamma \mid\!- e_1 : T_1 \quad \ldots \quad \Gamma \mid\!- e_n : T_n \quad |\{l_1, \ldots, l_n\}| = \mathsf{n}}{\Gamma \mid\!- \langle l_1 : e_1, \ldots, l_n : e_n \rangle : (+1 \mid \ell_1 : T_1, \ldots, \ell_n : T_n)}\ \text{T-STRC2}$$

$$\frac{\Gamma \mid\!- e_1 : T_1 \quad \ldots \quad \Gamma \mid\!- e_n : T_n \quad |\{\ell_1, \ldots, \ell_n\}| = \mathsf{n}}{\Gamma \mid\!- \langle\!\langle l_1 : e_1, \ldots, l_n : e_n \rangle\!\rangle : (0 \mid \ell_1 : T_1, \ldots, \ell_n : T_n)}\ \text{T-STRC1}$$

$$\frac{\Gamma \mid\!- \mathsf{e} : \mathsf{T} \quad (T, \ell) \in \mathrm{dom}(\mathrm{proj})}{\Gamma \mid\!- \mathsf{e}.\ell : \mathrm{proj}(S, \ell)}\ \text{T-PROJ}$$

$$\frac{\dfrac{}{\Delta \mid\!- \mathit{Number} : *}\ \text{K-NUM} \qquad \mathsf{x} \in \mathbb{Q} \setminus \mathbb{Z}}{\Gamma \mid\!- \mathsf{x} : \mathit{Number}}\ \text{T-NUMLIT}$$

$$\frac{\dfrac{}{\Delta \mid\!- \mathit{Null} : *}\ \text{K-NULL}}{\Gamma \mid\!- \mathsf{null} : \mathit{Null}}\ \text{T-NULL}$$

$$\frac{\Gamma \mid\!- e_1 \colon T_1 \quad \ldots \quad \Gamma \mid\!- e_n \colon T_n \quad \mathsf{T} \coloneqq T_1 \sqcup \ldots \sqcup T_n}{\Gamma \mid\!- [e_1, \ldots, e_n] \colon \mathsf{List\ T}}\ \text{T-LIST2}$$

$$\frac{}{\Gamma \mid\!- [\,] \colon \mathsf{List}\ \hat{\alpha}}\ \text{T-LIST1}$$

$$\frac{\dfrac{}{\Delta \mid\!- \mathit{Integer} : *}\ \text{K-INT} \qquad \mathsf{n} \in \mathbb{Z}}{\Gamma \mid\!- \mathsf{n} \colon \mathit{Integer}}\ \text{T-INTLIT}$$

$$\frac{\Gamma \mid\!- \mathsf{e} \colon \mathsf{T} \cap \mathsf{U} \qquad \dfrac{}{(T \cap U) <: \mathsf{U}}\ \text{S-INTER2} \qquad \dfrac{\Gamma \mid\!- \mathsf{e} \colon \mathsf{T} \qquad \mathsf{T} <: \mathsf{U}}{\Gamma \mid\!- \mathsf{e} \colon \mathsf{U}}\ \text{T-SUB}}{\Gamma \mid\!- \mathsf{e} \colon \mathsf{U}}\ \text{T-INTER3}$$

$$\frac{\Gamma \mid\!- \mathsf{e} \colon \mathsf{T} \cap \mathsf{U} \qquad \dfrac{}{(T \cap U) <: \mathsf{T}}\ \text{S-INTER1} \qquad \dfrac{\Gamma \mid\!- \mathsf{e} \colon \mathsf{T} \qquad \mathsf{T} <: \mathsf{U}}{\Gamma \mid\!- \mathsf{e} \colon \mathsf{U}}\ \text{T-SUB}}{\Gamma \mid\!- \mathsf{e} \colon \mathsf{T}}\ \text{T-INTER2}$$

$$\frac{\Gamma \mid\!- \mathsf{e} \colon \mathsf{T} \qquad \Gamma \mid\!- \mathsf{e} \colon \mathsf{U}}{\Gamma \mid\!- \mathsf{e} \colon \mathsf{T} \cap \mathsf{U}}\ \text{T-INTER1}$$

$$\frac{\hat{\alpha} \notin \mathsf{FV}(\Delta)}{\Delta, \hat{\alpha} \colon * \mid\!- \mathsf{wf}}\ \text{T-INF}$$

$$\frac{\dfrac{}{\Delta \mid\!- \mathit{Boolean} : *}\ \text{K-BOOL}}{\Gamma \mid\!- \mathsf{false} \colon \mathit{Boolean}}\ \text{T-FALSE}$$

$$\frac{\Gamma \mid\!- k_1 \colon \mathsf{K} \quad \ldots \quad \Gamma \mid\!- k_n \colon \mathsf{K} \quad \Gamma \mid\!- v_1 \colon V_1 \quad \ldots \quad \Gamma \mid\!- v_n \colon V_n \quad \mathsf{V} \coloneqq V_1 \sqcup \ldots \sqcup V_n}{\Gamma \mid\!- \{k_1 : v_1, \ldots, k_n : v_n\} \colon \mathsf{Dict\ K\ V}}\ \text{T-DICT2}$$

$$\frac{}{\Gamma \mid\!- \{\,\} \colon \mathsf{Dict}\ \hat{\alpha}\ \hat{\beta}}\ \text{T-DICT1}$$

$$\frac{\Gamma \mid\!- \mathsf{e} \colon \mathsf{T} \qquad \mathsf{T} <: \mathsf{U}}{\Gamma \mid\!- \mathsf{as}(e, U) \colon \mathsf{U}}\ \text{T-AS}$$

$$\frac{\Gamma \mid\!- \mathsf{f} \colon \mathsf{A} \to \mathsf{B} \qquad \Gamma \mid\!- \mathsf{a} \colon A' \qquad A' <: \mathsf{A}}{\Gamma \mid\!- (f\,a) \colon \mathsf{B}}\ \text{T-APP}$$

$$\frac{\Gamma, \mathsf{x} \colon \mathsf{A} \mid\!- \mathsf{e} \colon \mathsf{B}}{\Gamma \mid\!- \lambda\, \mathsf{x}.\ \mathsf{e} \colon \mathsf{A} \to \mathsf{B}}\ \text{T-ABS}$$

D.7. Generic

$$\frac{\Delta \mid\!- \mathsf{T} \colon *}{\Delta, \mathsf{T} <: \alpha \mid\!- \mathsf{wf}}\ \text{G-UPPER}$$

$$\frac{\Delta \mid\!- \mathsf{T} \colon *}{\Delta, \alpha <: \mathsf{T} \mid\!- \mathsf{wf}}\ \text{G-LOWER}$$

$$\frac{\Delta \mid\!- \mathsf{T} \colon *}{\Delta, \alpha = \mathsf{T} \mid\!- \mathsf{wf}}\ \text{G-EQUAL}$$

D.8. Quantification

$$\frac{T_1 <: T_2}{(\forall \alpha <: U.T_1) <: (\forall \alpha <: U.T_2)}\ \text{Q-SUB}$$

$$\frac{\Delta;\ \Gamma\ |-\ e: \tau \qquad \alpha \notin FV(\tau)}{\Delta;\ \Gamma\ |-\ e: \forall\ \alpha <: \top.\ \tau}\ \text{Q-GEN}$$

$$\frac{\Delta, U: *\ |-\ T: K \qquad \beta \notin FV(T)}{\Delta\ |-\ \forall\ \alpha <: U\ .\ T == \forall\ \beta <: U\ .\ T[\alpha \mapsto \beta]}\ \text{Q-CONV}$$

$$\frac{\Delta;\ \Gamma\ |-\ e: \forall\ \alpha <: U.\ \tau \qquad \Delta\ |-\ S <: U}{\Delta;\ \Gamma\ |-\ e\ [\![S]\!]: \tau[\alpha \mapsto S]}\ \text{Q-APP}$$

D.9. Recursion

$$\frac{}{\Delta\ |-\ \mu\ X.\ T \triangleq T[X \mapsto \mu X.T]}\ \text{R-UNFOLD}$$

D.10. Union

$$\frac{}{T \cup \top = \top}\ \text{U-TOP}$$

$$\frac{\dfrac{S_1 <: T_1 \qquad \dfrac{S <: T}{S <: (T \cup U)}\ \text{S-UNION3}}{S_1 <: (T_1 \cup T_2)} \qquad \dfrac{S_2 <: T_2 \qquad \dfrac{S <: U}{S <: (T \cup U)}\ \text{S-UNION2}}{S_2 <: (T_1 \cup T_2)} \qquad \dfrac{T <: S \qquad U <: S}{(T \cup U) <: S}\ \text{S-UNION1}}{(S_1 \cup S_2) <: (T_1 \cup T_2)}\ \text{U-MONO}$$

$$\frac{\dfrac{\dfrac{}{U <: U}\ \text{S-REFL} \qquad \dfrac{S <: U}{S <: (T \cup U)}\ \text{S-UNION2}}{U <: (T \cup U)} \qquad \dfrac{T <: U \qquad U <: V}{T <: V}\ \text{S-TRANS}}{U <: S}\ \text{U-INV2}$$

$$\frac{\dfrac{\dfrac{}{T <: T}\ \text{S-REFL} \qquad \dfrac{S <: T}{S <: (T \cup U)}\ \text{S-UNION3}}{T <: (T \cup U)} \qquad \dfrac{T <: U \qquad U <: V}{T <: V}\ \text{S-TRANS}}{T <: S}\ \text{U-INV1}$$

$$\frac{}{T \cup T = T}\ \text{U-IDEM}$$

$$\frac{}{T \cup (U \cap V) = (T \cup U) \cap (T \cup V)}\ \text{U-DIST}$$

$$\frac{}{S \cup T = T \cup S}\ \text{U-COMM}$$

$$\frac{}{T \cup \bot = T}\ \text{U-BOT}$$

$$\frac{}{(S \cup T) \cup U = S \cup (T \cup U)}\ \text{U-ASSOC}$$

D.11. Intersection

$$\frac{}{\mathsf{T} \cap \top = \mathsf{T}}\ \text{I-TOP}$$

$$\frac{\dfrac{}{(S_1 \cap S_2) <: S_1}\ \text{S-INTER1} \quad S_1 <: T_1}{(S_1 \cap S_2) <: T_1}\ \text{S-TRANS} \qquad \frac{\dfrac{}{(S_1 \cap S_2) <: S_2}\ \text{S-INTER2} \quad S_2 <: T_2}{(S_1 \cap S_2) <: T_2}\ \text{S-TRANS}$$

$$\frac{(S_1 \cap S_2) <: T_1 \qquad (S_1 \cap S_2) <: T_2}{(S_1 \cap S_2) <: (T_1 \cap T_2)}$$

$$\frac{\mathsf{S} <: (T \cap U) \qquad \dfrac{}{(T \cap U) <: \mathsf{U}}\ \text{S-INTER2} \qquad \dfrac{\mathsf{T} <: \mathsf{U} \qquad \mathsf{U} <: \mathsf{V}}{\mathsf{T} <: \mathsf{V}}\ \text{S-TRANS}}{\mathsf{S} <: \mathsf{U}}\ \text{I-INV2}$$

$$\frac{\mathsf{S} <: (T \cap U) \qquad \dfrac{}{(T \cap U) <: \mathsf{T}}\ \text{S-INTER1} \qquad \dfrac{\mathsf{T} <: \mathsf{U} \qquad \mathsf{U} <: \mathsf{V}}{\mathsf{T} <: \mathsf{V}}\ \text{S-TRANS}}{\mathsf{S} <: \mathsf{T}}\ \text{I-INV1}$$

$$\frac{}{\mathsf{T} \cap \mathsf{T} = \mathsf{T}}\ \text{I-IDEM}$$

$$\frac{}{\mathsf{T} \cap (U \uplus V) = (T \cap U) \uplus (T \cap V)}\ \text{I-DIST}$$

$$\frac{}{\mathsf{S} \cap \mathsf{T} = \mathsf{T} \cap \mathsf{S}}\ \text{I-COMM}$$

$$\frac{}{\mathsf{T} \cap \bot = \bot}\ \text{I-BOT}$$

$$\frac{}{(S \cap T) \cap \mathsf{U} = \mathsf{S} \cap (T \cap U)}\ \text{I-ASSOC}$$

D.12. Extrema

$$\frac{\mathsf{S} \coloneqq T_0 \times \ldots \times T_{n-1} \qquad \bot \in \mathrm{rng}(S)}{\mathsf{S} == \bot}\ \text{E-TUP}$$

$$\frac{\mathsf{S} \coloneqq (\omega \mid \ell_1 : T_1, \ldots, \ell_n : T_n) \qquad \bot \in \mathrm{rng}(S)}{\mathsf{S} == \bot}\ \text{E-STRC}$$

$$\frac{\mathsf{N} \coloneqq o\langle s | \bot \rangle}{\mathsf{N} = \bot}\ \text{E-NOM}$$