Lento: A Readable, Safe, and Efficient Language

[DRAFT 2026-01-25]

William Rågstad
Independent Researcher

***Abstract*—The result of decades of programming language research can become diluted when retrofitted into mainstream languages. Legacy constraints and compatibility requirements force compromises that lead to complex and unwieldy designs. Consequently, redesigning a new modern language from the ground up is often necessary to fully leverage state-of-the-art advances in typing, verification, and compilation. In this paper we introduce a *high-level and work-in-progress vision* of Lento, a language designed to bridge the gap between high-level abstraction and low-level performance.**

***Index Terms*—Programming Language, Parallelism, Guided Synthesis, Program Specification, Formal Verification, Liquid Types, Dependent Types, Session Types, Effect System, Type Inference**

I. Introduction

Programming languages today often force developers to choose between high-level abstractions that enhance productivity and low-level control that optimizes performance. This dichotomy has led to a fragmented landscape where **languages excel in one domain but falter in the other**. High-level languages, such as Python and JavaScript, prioritize ease of use and rapid development but often **sacrifice performance** and compile-time **type safety**. Conversely, low-level languages like C and Rust offer fine-grained control over system resources but come with **steep learning curves**, **increased complexity**, and **higher demands** on expertise.

Lento aims to bridge this gap by introducing a novel programming language while also incorporating the benefits of the pure functional paradigm. Leveraging strong static typing, lazy evaluation, automated parallelism and concurrency, user-friendly syntax, and more, to offer several advantages over existing languages. It is designed to be **safe**, **performant**, and **stable** while promoting **longevity** in codebases.

II. Values

In the Lento compiler team we define a fundamental set of values people developing the language should adhere to. These extend to the language standard specification and form the basis for all future features and decisions. Our standpoints on these values are outlined below and are goals we strive to achieve in the development of Lento.

A. Longevity

Lento should be a language built to last for decades. We have learned from the mistakes of others and decide to support multiple versions of the language (called language editions[1]) simultaneously. Where libraries in one edition must be backward compatible and seamlessly interoperate with those compiled in other editions. Editions with major breaking changes should include migration guidelines. This allows users to choose when to upgrade and ensures that older codebases remain functional with long-term support (LTS).

B. Stability

Lento should prioritize stability in its features, standard library APIs, and language design.

Once a feature is released, it should not change in a way that breaks existing code. To achieve this, rigorous reviewing of new features include both community feedback and thorough testing. Most importantly, features are designed with future compatibility in mind from the start. Defining a small "external" feature interface to the language, allowing internal changes with semantically equivalent behavior.

C. Performance

Lento strive to be a high-performance language in terms of execution speed and resource efficiency. However, our goal is not to compete with other low-level languages, but rather to provide predictable performance characteristics.

D. Bootstrapping

One controversial standpoint and conscious design choice is that Lento should not be bootstrapped in itself. Instead, we believe that an implementation written in the low-level systems language Rust is a better choice for achieving our goals for the near future. This enables faster and more flexible development between major language editions while still allowing for future self-hosting when Lento is matured. Rust offers the Lento compiler a solid foundation for performance, safety, and cross-platform compatibility.

E. Safety

Lento is designed with safety as a core principle. This includes memory safety, type safety, concurrency safety and behavioral safety through formal program specifications. By leveraging a strong static type system with dependent liquid refinements and a static effect-system, Lento aims to

[1]This is similar to Rust's approach with new editions every 3 years, see https://blog.rust-lang.org/2014/10/30/Stability.

catch errors at compile-time, reducing runtime failures and ensuring that programs behave as intended. Lento builds on top of the principle of immutability as a cornerstone of functional programming, making programs easier to reason about and debug. Mutability and side effects are handled safely while maintaining referential transparency.

F. Interoperability

All editions of Lento must be backwards compatible with each other at the binary level through a stable Application Binary Interface (ABI). Two programs compiled with different editions of Lento must be able to interoperate without issues.

Lento should also prioritize interoperability with existing languages and ecosystems. This includes providing a robust Foreign Function Interface (FFI) and ABI marshalling for seamless integration with languages such as C/C++, Rust, and Python.

G. Tooling and Ecosystem

Lento should provide a comprehensive tooling ecosystem to enhance developer productivity. This includes a package manager for easy dependency management, IDE integrations for code completion and debugging, and build tools for efficient project management.

H. Community

Our final standpoint is that we should foster a welcoming and inclusive community around Lento that encourages constructive collaboration and knowledge sharing. This involves creating clear contribution guidelines, maintaining open communication channels, and actively engaging with users and contributors to gather feedback and improve the language.

III. Features

Lento aims to provide a robust set of features that enhance developer productivity while maintaining the core values of the language.

A. Strong Static Type System

This is a fundamental feature and core value of the language itself. Lento provides a dependent type system with liquid refinements and effects to catch both *regular type errors* and *logical contradictions* in specifications all at compile-time using bidirectional type inference without sacrificing expressivity. See Section V for a more detailed explanation of Lento's type system.

B. Automated Parallelism

Lento targets scalable task-level parallelism with automatic management of parallel work granularity and scheduling, informed by recent advances in automatic parallelism management [1] and classic approaches to automatic parallelization [2].

C. Concurrency Model

D. Memory Model

Region-based memory management and memory safety without garbage collection via linear/affine types, ownership, and borrowing.

E. Program Synthesis

Lento incorporates program synthesis techniques to automatically generate code snippets based on type-level program specifications provided by the developer, see Section V.G. This aligns with type-driven synthesis from polymorphic refinement types [3].

F. Multistage Meta-Programming

Lento supports multistage meta-programming for constant evaluation and staging code generation at compile-time using a staged calculus in a type-safe way with phase separation that is more powerful than traditional hygienic macros.

G. Partial Evaluation

Lento supports partial evaluation to optimize programs by precomputing parts of the code at compile-time based on known inputs and contexts, resulting in more efficient runtime performance. This is *similar to multistage programming* but focuses on optimization rather than code generation, see Section III.F.

H. Holes

Lento supports the use of holes in code, allowing developers to leave parts of the program unspecified during rapid development. Instead of employing gradual typing, holes in Lento must be statically typed but can be left unimplemented. This marks a place for the compiler to "fill in" an implementation using program synthesis techniques (see Section III.E) based on the surrounding context, type information, and specifications.

I. Standard Library

Lento should provide two sets of *officially-supported* standard libraries. First a stable cross-platform core, and extended ext libraries with wide range of functionality out of the box[2]. The extended libraries may have platform-specific dependencies and less stability guarantees compared to the core libraries. See Fig. 1 for the standard library lattice in Lento.

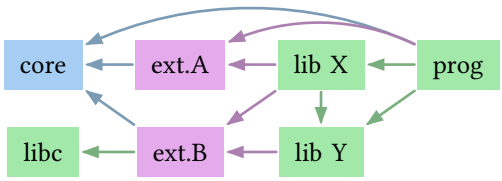

Fig. 1. Standard library hierarchy with core core in blue, extended ext libraries in purple, and user-defined packages in green.

[2]This is heavily inspired by both the Rust and Go standard library designs, see https://doc.rust-lang.org/std and https://pkg.go.dev/std.

a) *Core core:* Essential data structures (lists, maps, sets, graphs), algorithms (sorting, searching), concurrency (async/await), networking (TCP, UDP), and file I/O APIs.

b) *Extended ext:* Additional libraries and frameworks for common tasks and domains, such as OS interaction (Win32, POSIX), web client and server development (HTTP, WebSocket), data processing (CSV, JSON), databases (SQL, NoSQL), GUI toolkits, scientific computing (numerical methods, statistics), computer graphics (2D/3D rendering, game development), compression (gzip, zlib), cryptography (hashing, encryption), machine learning (linear algebra, neural networks), and more.

c) *Third-party libc:* Lento does not aim to "reinvent the wheel", but use what knowledge and tools are available. One such resource is libc, a powerful and backwards compatible API with highly efficient implementations. The libc API is cross-platform, offered via GNU och MUSL on Linux, and UCRT on Windows. This library is a trusted base to build on top of with the benefits of:

- Less library code in Lento thanks to reusability.
- Dynamic linking results in light-weight artifacts.
- Performant runtime via optimized implementations.

J. Tooling Ecosystem

A language is only as good as its tooling and ecosystem. Lento should therefore provide a comprehensive developer tooling to enhance productivity and streamline processes.

a) *Package Management:* Lento should provide a built-in package manager to facilitate easy dependency management, versioning, and distribution of libraries and applications. This includes support for semantic versioning, dependency resolution, and publishing packages to an official default central repository configurable by the user. Self-hosted registries should also be supported for private proprietary use cases.

b) *IDE Integrations:* LSP support for popular text editors to provide code completion, syntax highlighting, inline error diagnostics, and debugging capabilities.

c) *Build Tools:*

d) *Testing Framework:*

e) *Documentation Generation:* The language should include tools for generating comprehensive documentation from function signatures, but also from code comments and type annotations. Comments should support markdown-like syntax for formatting, code snippets, and linking to other parts of the documentation. The primary source of truth should always be type signatures to ensure accuracy and consistency thanks to Lento's strong typing with dependent refinements and formal specifications.

IV. Language Syntax

This section presents the formal syntax of expressions, types, and program structure.

A. Lento by Example

We begin with a brief overview of Lento's syntax via some examples. Consider the following recursive fact function:

```
1  fact(n: int, acc = 1) : int = match n                    Lento
2      | 0 => acc,
3      | _ => fact(n - 1, n * acc),
```

Here, we define a function fact that takes an integer n and returns its fact. The match expression allows us to pattern match on the value of n, providing different cases for 0 and other integers.

We could also define the type of fact using dependent types and liquid refinements to specify that the result is always non-negative and utilize pattern matching in argument bindings [4]:

```
1  type Nat = { v: int | v >= 0 }                           Lento
2  fact :: Nat -> Nat -> Nat
3  fn fact(n, acc) = fact(n - 1, n * acc)
4  fn fact(0, acc) = acc
```

Note that the function clause order does not matter *here*, due to automatic ordering of case specificity placing the general case last for us. This is usually done automatically by the compiler, but for ambiguous cases, the compiler requires us to specify the order ourselves explicitly.

B. Formal Syntax

Language terms are defined by the following expression grammar:

$$
\begin{aligned}
e &::= cnst \mid stm \mid e_1\ e_2 \mid (p : \tau) \Rightarrow e \\
&\mid \oplus e \mid e_1 \oplus e_2 \mid e \oplus \mid \{e_1, ..., e_n\} \\
&\mid [e_1, ..., e_n] \mid \{l_i : e_i\} \mid (e_1, ..., e_n) \\
&\mid T(e_1, ..., e_n) \mid (e : \tau) \\
cnst &::= () \mid n \mid n.n \mid true \mid false \mid \text{"s"} \mid \text{'c'} \\
stm &::= \text{id} :: \tau \\
&\mid \textbf{let}\ p = e_1\ \text{in}\ e_2 \\
&\mid \textbf{type}\ t = \tau\ \text{in}\ e \\
&\mid \textbf{fn}\ f(p_1, ..., p_n) : \tau = e_1\ \text{in}\ e_2 \\
&\mid \textbf{match}\ e\ \text{with}\ \{p_1 \Rightarrow e_1, ..., p_n \Rightarrow e_n\} \\
&\mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \\
&\mid \textbf{import}\ m\ \text{in}\ e \\
p &::= cnst \mid \text{id} \mid \_ \mid p : \tau \mid \oplus p \mid p_1 \oplus p_2 \mid p \oplus \\
&\mid [p_1, ..., p_n] \mid \{l_i : p_i\} \mid (p_1, ..., p_n) \\
&\mid T(p_1, ..., p_n) \\
t &::= \text{id} \mid \textbf{forall}\ \text{id} : \tau\ \text{in}\ t \mid T(\text{id}_1, ..., \text{id}_n) \\
m &::= \text{id} \mid \text{id}\ .\ m \mid \text{id}_1\ \textbf{as}\ \text{id}_2 \mid \{m_1, ..., m_n\}
\end{aligned}
\tag{1}
$$

Both n and i are metavariables ranging over numeric literals and identifiers, respectively. The constants s and c represent arbitrary string and character literals. The symbol τ ranges over types and T type constructors such as $Some(e)$ of type Option τ where $e : \tau$, defined in Section V.A. The symbol $\oplus$ ranges over operators (arithmetic, logical, comparison, etc.) in different fixity positions (prefix, infix, postfix). Statements in stm use standard keywords such as **let**, **fn**, **match**, and **if** that are entirely syntactic sugar and desugar to core expressions in e. Only **match** is

a fundamental construct in the operational semantics, see Section VI. The formal "in e" syntax is *implicit* and only included for clarity.

V. Type System

The design unifies several advanced features within its strong static type system and algebraic effect system to catch a wide range of logical errors at compile time. The type syntax is expressive yet easy to use due to our inference-first principle, allowing developers to define complex types and guarantee properties about their program behavior while minimizing boilerplate code. This chapter describes a formal operational semantics defining program execution: surface-language constructs elaborate into a typed core calculus in which typing and refinements are checked. Type aliases are supported as syntactic sugar but do not impact the core type system.

A. Type Syntax

The core type language is defined inductively for any type-level term τ as follows:

$$
\begin{aligned}
\tau &::= A \mid B \mid C \mid D \mid E \mid M \mid L \mid S \mid N \mid \mathbf{type}_i \\
A &::= \tau_i \to \tau_j \mid \tau_i\ \tau_j \\
B &::= \text{unit} \mid \text{bool} \mid \text{char} \mid \text{str} \mid \text{uint}_u \mid \text{int}_w \mid \text{float}_v \mid \text{num} \\
C &::= \text{record}\{\ell_i : \tau_i\} \mid \text{inductive}\{\ell_i : \tau_i\} \\
D &::= \Pi x : \tau_i.\tau_j \mid \Sigma x : \tau_i.\tau_j \\
E &::= x : \tau_i \to \varepsilon \to \tau_j \mid \tau_i\ !\ \varepsilon \to \tau_j \mid \tau_i\ ?\ \varepsilon \to \tau_j \\
M &::= \Box\tau \mid \Diamond\tau \\
L &::= \{\nu : \tau \mid \varphi\} \\
S &::= \mathbf{end} \mid\ !\tau.S \mid ?\,\tau.S \mid \mu t : S \mid \oplus\{\ell_i : S_i\} \mid \&\{\ell_i : S_i\} \\
N &::= \text{literal}(c) \\
\varphi &::= \text{true} \mid \text{false} \mid \nu \oplus c \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi
\end{aligned}
\tag{2}
$$

Note that $i, j \in I$ ranges over indices in finite index sets I, $u \in \{1, ..., 128, \text{big}\}$ and $w \in \{8, ..., 128, \text{big}$ denotes signed/unsigned integer bit-widths, $v \in \{32, 64, \text{big}\}$ denotes a floating-point bit-width. Literal constants c narrow types of category B. Predicates φ range over an SMT-decidable logic, $\oplus$ abbreviates a comparison operator (e.g., $=, \neq, <, \leq, >, \geq$) against a constant c. The refinement variable ν ranges over the value denoted by a refined term. Type categories are as follows:

- A: Type constructor application of parameterized types and type-level functions (e.g., Map τ_i τ_j).
- B: Primitive base types and literals.
- C: Composite types (records and recursive sum types with type constructors).
- D: Dependent types.
- E: Effect-annotated function types.
- M: Modal types.
- L: Liquid refinement types.
- S: Session types.
- N: Literal singleton types (for constant propagation and partial evaluation), e.g literal(42).
- $\mathbf{type}_i$: The universe of types at level $i \in \mathbb{N}$ (to support cumulative type hierarchies and avoid Girard's paradox).

The standard library provides numerous predefined types and type constructors built on top of this core syntax, see Section III.I. These include common data structures (Tuple τ_i ... τ_j, Vec ν τ, Map τ_i τ_j, Set τ), parameterized polymorphic algorithms (sorting, searching), concurrency abstractions (async/await effects), and networking primitives (TCP/UDP channels with session types). There is also a type class $\mathbf{Chan}(S)$ for abstracting over communication channels (see Section V.E).

Example composite types in C are instances of record types and inductive types (algebraic data types) such as tuples, lists, 2D points, and option types:

$$
\begin{aligned}
&\text{Tuple} ::= \text{record}\{0 : \text{int}, 1 : \text{bool}, 2 : \text{float}\} \\
&\text{List}\ \tau ::= \text{inductive}\{\mathit{Nil} : \text{unit}, \\
&\qquad\qquad \mathit{Cons} : \Sigma\ \text{head} : \tau.\ \text{List}\} \\
&\text{Point2D} ::= \text{record}\{x : \text{int}, y : \text{int}\} \\
&\text{Option}\ \tau ::= \text{sum}\{\mathit{Some} : \tau, \mathit{None} : \text{unit}\}
\end{aligned}
\tag{3}
$$

Types in D include dependent function types (Π types) and dependent pair types (Σ types). Algebraic effect-annotations ε denote function side-effects in category E, while refined types are in L (see Section V.C) and session types are in S (see Section V.E).

B. Dependent Types

Types can be parameterized by values to encode rich invariants and relationships. This enables the expression of rich invariants and relationships between data directly in the type system. For example, we can define a vector type parameterized by its length:

$$\text{Vec}(\nu : \text{Int}, \tau : \text{Type}) = \{\nu : \text{List}(\tau) \mid \text{len}(\nu) = \nu\} \tag{4}$$

Here, $\text{Vec}(n, \tau)$ represents a list of elements of type τ with length exactly n. This permits specifications such as length-indexed collections, value-range invariants, and relational postconditions, while still admitting decidable checking once refinements are restricted to an SMT-decidable fragment. Foundational accounts of dependent and advanced types can be found in [5] and [6].

C. Liquid Refinement Types

Liquid refinements qualify base types with logical predicates, yielding logically qualified data types. Refined base types have the form $\{\nu : B \mid \varphi\}$, where B is a base type and φ is an SMT-decidable predicate verifying properties of the value ν at compile-time. Unless otherwise stated, predicates are interpreted under the variable assignments induced by Γ. For example, the type $\{\nu : \text{Int} \mid 0 \leq \nu \wedge \nu < n\}$ denotes integers in the half-open interval $[0, n)$. To preserve decidability, the refinement logic is restricted to an SMT-decidable subset (typically a quantifier-free fragment), including theories such as linear arithmetic, uninterpreted functions, and finite maps. See [4] for the original development and [7] for subsequent extensions.

D. Modal Types

Lento incorporates modal type constructors to model staged and time-dependent computation. We write necessity and possibility modalities as $\Box\tau$ and $\Diamond\tau$ respectively. Intuitively, $\Diamond\tau$ classifies computations that may produce a τ in the future. These modalities interact with staged meta-programming (see Section III.F), effect tracking (see Section V.F), concurrency abstractions (see Section III.C), and parallelism (see Section III.B). For example, asynchronous functions can be assigned modal signatures that make temporal availability explicit:

$$\begin{aligned} &\text{async fetch}(\text{url} : \mathit{String}) \to \Diamond\ \mathit{Data} \\ &\text{await} : \Diamond\ \mathit{Data} \to \mathit{Data} \end{aligned} \tag{5}$$

Here $\Diamond\ \mathit{Data}$ records that the result is not necessarily available required by the temporal aspect of this particular asynchronous computational effect.

E. Session Types

Lento uses session types to specify and enforce structured communication protocols. Channels $\mathbf{Chan}(S)$ are parameterized by a session type S to statically guarantee correct sequences of message passing between two parties. A session type consists of special type operators modeling sending and receiving messages, branching, selection, and recursion. The underlying implementation of channels can be separated from the session type specification, allowing for various transport mechanisms (e.g., TCP, in-memory queues), optimizations (e.g., zero-copy), or serialization formats (e.g., JSON, Protobuf[3]). We write output and input actions as $!\tau.S$ and $?\,\tau.S$, respectively, together with external choice (branching) and internal choice (selection):

$$\begin{aligned} S ::= \ &\mathbf{end} \mid t \mid\ !\tau : S \mid ?\,\tau : S \mid \mu\tau : S \\ &\mid\ \&\{\ell_i : S_i\}_{\{i\in I\}} \mid \oplus\{\ell_i : S_i\}_{\{i\in I\}} \end{aligned} \tag{6}$$

Where $\mathbf{end}$ denotes the end of a session and t a basic type in B, see Section V.A. For real distributed systems with multiparty sessions (MST) we can use the global protocol type G to describe the overall communication protocol among multiple participants, and project it onto local session types $S_p = G \downarrow p$ for each participant p to ensure that each party adheres to the global protocol. This guarantees communication safety and protocol compliance by construction, with automatic duality between send and receive operations (client/server) by inverting sends/receives and select/branch direction, denoted $\overline{S}$.

$$\begin{aligned} \overline{!\tau.S} &= ?\,\tau.\overline{S} \qquad \overline{?\,\tau.S} = !\tau.\overline{S} \qquad \overline{\mu t.S} = \mu t.\overline{S} \\ \overline{\oplus\{\ell_i : S_i\}_{\{i\in I\}}} &= \&\left\{\ell_i : \overline{S_i}\right\}_{\{i\in I\}} \\ \overline{\&\{\ell_i : S_i\}_{\{i\in I\}}} &= \oplus\left\{\ell_i : \overline{S_i}\right\}_{\{i\in I\}} \end{aligned} \tag{7}$$

[3]Protocol Buffers are language-neutral, platform-neutral extensible mechanisms for serializing structured data. From https://protobuf.dev

F. Effect System

Lento incorporates static algebraic liquid effect system to reason about and handle side effects in programs such as I/O operations, state mutations, exceptions, concurrency, and even deterministic parallelism. [8] This allows developers to reason about the effects their code may have, enabling safer and more predictable programming patterns.

G. Program Specifications

Verifiable type annotations and function signatures express preconditions, postconditions, and invariants are used to reason about program behavior statically and formally to ensure runtime safety. This allows developers to write more reliable code and reduce the likelihood of logical bugs.

H. Syntax and Grammar

The core type language τ is defined inductively. We distinguish between base types B (primitive scalars defined in the lattice) and refined types.

$$\begin{aligned} \tau &::= B \mid \{\nu : B \mid \varphi\} \mid x : \tau_1 \to \varepsilon \to \tau_2 \\ B &::= \text{Unit} \mid \text{Int}_w \mid \text{UInt}_w \mid \text{Float}_w \mid \text{Bool} \\ \varphi &::= \text{true} \mid \text{false} \mid \nu \oplus c \mid \varphi_1 \wedge \varphi_2 \end{aligned} \tag{8}$$

Where $w \in \{1, \ldots, 128, \text{big}\}$ represents the bit-width of the integer, and ε represents the effect set associated with function execution. The logical predicates φ are expressed in an SMT-decidable fragment, allowing for automated verification during type checking where $\oplus$ represents equality or inequality to constant c such as $\nu = 42$, $\nu \neq 0$ or $\nu > 0$.

a) *Examples:*

```
1  // Example of user-defined types                    Lento
2  Age = uint
3  Point = { x: float, y: float }
```

I. The Numeric Lattice and Subtyping

Lento supports a subtyping mechanism that allows for hierarchical relationships between types. This enables polymorphism and code reuse, as functions can accept arguments of a supertype while operating on subtypes.

Unlike standard systems where numeric types are disjoint, Lento organizes scalars into a bounded lattice structure $\mathcal{L} = (B, \subseteq)$. Subtyping is defined via lattice inclusion rather than implicit coercion, simplifying the elaboration phase.

We define the subtyping relation $<:$ as the conjunction of structural inclusion and logical implication.

$$\frac{B_1 \subseteq B_2 \qquad \text{Valid}_{\text{SMT}}(\varphi_1 \Rightarrow \varphi_2)}{\Gamma \vdash \{\nu : B_1 \mid \varphi_1\} <: \{\nu : B_2 \mid \varphi_2\}}\ \text{Sub-Refine} \tag{9}$$

Fig. 2. Refinement Subtyping Rule. A type is a subtype if its base is smaller in the lattice AND its logical predicate implies the parent's predicate.

This allows strictly checked narrow types (e.g., **u8**) to be used where wider types (e.g., **u64**) are expected, but not vice versa, without runtime casting.

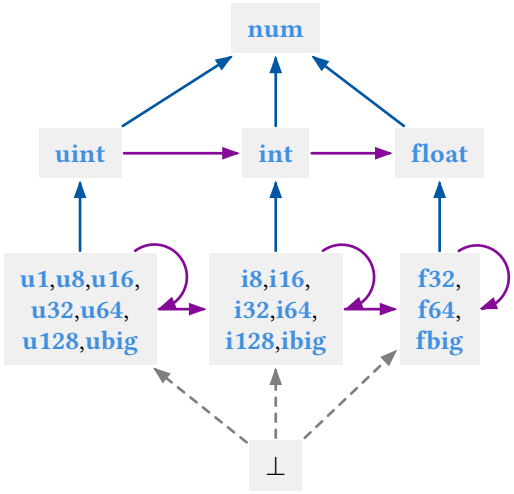

Fig. 3. Numeric type lattice with subtyping in blue and widening in purple.

See Fig. 3 for the number subtyping lattice in Lento illustrating primitive widening type conversions (numeric promotion) and subtyping relationships. [9]

J. Bidirectional Type Inference

Lento employs a bidirectional type inference algorithm num that combines both type checking and type synthesis. This approach allows the compiler to infer types of expressions without requiring explicit type annotations.

To support local type inference without global unification (which is undecidable with dependent refinements), Lento utilizes a bidirectional discipline. We split the typing judgment into two modes: Synthesis ($\Longrightarrow$) and Checking ($\Longleftarrow$).

a) *Synthesis ($\Longrightarrow$):* In synthesis mode, the compiler generates a type τ for an expression e based on the environment Γ.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \{\nu : \tau \mid \nu = x\}} \text{ Var} \tag{10}$$

b) *Checking ($\Longleftarrow$):* In checking mode, the compiler verifies expression e against a known type τ. Crucially, this allows for "smart" literals that adapt to their expected width.

$$\frac{\Gamma \vdash e \Rightarrow \tau' \quad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash e \leq \tau} \text{ Subsumption} \tag{11}$$

K. Operational Semantics

Lento uses a call-by-value, big-step operational semantics. The core reduction rule for let-binding illustrates the flow-sensitive nature of the environment update.

$$\frac{\Gamma \vdash e_1 \Uparrow v_1 \quad \Gamma[x \mapsto v_1] \vdash e_2 \Downarrow v_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \Downarrow v_2} \text{ E-Let} \tag{12}$$

VI. Operational Semantics

This section defines the small-step operational semantics of Lento using a standard evaluation relation $e \to e'$ over expressions.

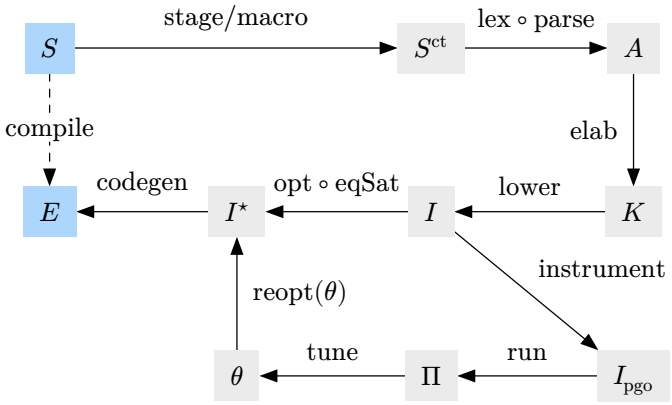

Fig. 4. Internal compilation pipeline with stages from source code to final artifact highlighted in blue

VII. Compilation

To effectively translate Lento's high-level abstractions and rich type system into efficient executable code, a sophisticated multistage compilation process is required.

A. Overview

Including staging, elaboration, lowering, optimization, code generation, and profiling with feedback for re-optimization.

Interpretation:

B. Objects (nodes)

- S: Source program
- S^{ct}: Staged/expanded source (after hygienic macros / MSP / const-eval)
- A: AST / syntax tree
- K: "Semantic IR" typed core calculus (post-elaboration, bidirectional typing, holes resolved/registered)
- I: Mid-level IR (e.g., SSA-ish or semantic dialect)
- $I^{\star}$: Optimized IR
- E: Final artifact (object file / executable / WASM module)
- $K_{\text{spec}}, I_{\text{spec}}$: Specialized (partial-evaluated) variants
- I_{pgo}: Instrumented IR
- Π: Profiling data
- θ: Optimization parameters / decisions derived from (Π)

C. Morphisms (edges)

- stage/macro: Hygienic macros, MSP phase splitting, const-eval fragment
- lex ∘ parse: Lexer+parser
- elab: Elaboration/type inference (bidirectional) into core
- lower: Lowering into IR
- opt: Standard optimizations + fusion + specialization triggers + etc.
- eqSat/rewrite: Equality saturation / rewrite-rule engine for library-driven optimization
- partialEval: Specialization w.r.t. known inputs
- instrument, run, tune: PGO loop with a "pause" awaiting external execution input

References

[1] S. Westrick, M. Fluet, M. Rainey, and U. A. Acar, "Automatic Parallelism Management," *Proc. ACM Program. Lang.*, vol. 8, no. POPL, Jan. 2024, doi: 10.1145/3632880.

[2] S. P. Midkiff, *Automatic Parallelization.* in Synthesis Lectures on Computer Architecture. Cham: Springer International Publishing, 2022. doi: 10.1007/978-3-031-01736-0.

[3] N. Polikarpova, I. Kuraj, and A. Solar-Lezama, "Program synthesis from polymorphic refinement types," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, in PLDI '16. Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pp. 522–538. doi: 10.1145/2908080.2908093.

[4] P. M. Rondon, M. Kawaguci, and R. Jhala, "Liquid types," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, in PLDI '08. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 159–169. doi: 10.1145/1375581.1375602.

[5] H. Xi and F. Pfenning, *Advanced Types in Programming Languages.* in Titolo collana. Cambridge University Press, 2016. [Online]. Available: https://dl.acm.org/doi/book/10.5555/1076265

[6] B. Pierce, *Types and Programming Languages.* in MIT Press Books. MIT Press, 2002. [Online]. Available: https://dl.acm.org/doi/book/10.5555/509043

[7] M. Woo-Kawaguchi, "High-Level Liquid Types," Doctoral dissertation, 2016. [Online]. Available: https://escholarship.org/uc/item/7d8525sz

[8] M. Kawaguchi, P. Rondon, A. Bakst, and R. Jhala, "Deterministic parallelism via liquid effects," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, in PLDI '12. Beijing, China: Association for Computing Machinery, 2012, pp. 45–54. doi: 10.1145/2254064.2254071.

[9] F. Pfenning, "Lecture Notes on Subtyping," 2023, [Online]. Available: https://www.cs.cmu.edu/~fp/courses/15836-f23//lectures/08-subtyping.pdf