PLSS 2025 – Programming Language Standardization and Specification Workshop Report

Mikhail Barash* **and Yulia Startsev†**

*Bergen Language Design Laboratory, University of Bergen, Norway

†Mozilla, Berlin, Germany

ABSTRACT This report presents a summary of the talks and discussions held at the Workshop on Programming Language Standardization and Specification (PLSS 2025). The workshop brought together language designers, implementers, researchers, and practitioners to examine how programming languages are specified, evolved, and standardized in practice. The talks explored a diverse set of programming languages, including WebAssembly, C++, JavaScript, Kotlin, Scheme, P4, APL, Emacs Lisp, and the new Hylo language. Several speakers addressed socio-technical aspects of programming language standardization, and emphasized the need for a better integration between the specification and the implementation processes.

KEYWORDS Language Evolution, Mechanization, Language Standardization, Language Specification, Formal Semantics.

1. Introduction

The evolution of widely adopted programming languages is critical for ensuring their sustainability, interoperability, and adaptability to changing technological and societal needs. The Workshop aimed to advance the understanding of programming language standardization and foster collaborative solutions for its challenges. Participants were given an opportunity to share insights, case studies, and best practices to shape the future of programming language specification and evolution.

The workshop examined the role of specifications as the foundation for standards documents, focusing on a wide range of topics, such as:

- mechanized specifications of programming languages
- formal methods in programming language specifications
- decision making in programming language standardization
- intellectual property rights issues in programming language standardization
- socio-technical aspects of programming language evolution
- implications for language adoption

JOT reference format:
Mikhail Barash and Yulia Startsev. *PLSS 2025 – Programming Language Standardization and Specification Workshop Report*. Journal of Object Technology. Vol. 25, No. 1, 2026. http://dx.doi.org/10.5381/jot.2026.25.1.e5

Across the diverse talks presented at the workshop, a number of recurring themes emerged, suggesting shared challenges in programming language design and standardization. Perhaps the most consistent theme was the emphasis on long-term evolution. Whether designing a new language, maintaining a widely-used legacy system, or creating a mechanized specification, speakers repeatedly emphasized the need to support software and language development over time. This included minimizing breaking changes, supporting modularity, and enabling backward-compatible extensions.

Modularity emerged as a recurring design ideal, often linked to the goal of facilitating maintainability and reducing coupling. This tied closely to another common thread: the importance of user feedback and practical applicability. Multiple talks stressed that programming languages are not theoretical artifacts, but tools used by real developers. Incorporating user needs—through tight feedback loops, test suites, or community engagement—was presented as essential for sustainable progress.

The contrast between incremental design and monolithic development was another shared point. Several speakers critiqued the drawbacks of long, isolated design cycles and instead endorsed agile, stepwise refinement. This applied both to implementation strategies and to standardization processes. In this context, several talks also confronted the inherent complexity of specification, particularly when prose documents diverge from

implementations or formal models.

Another significant shared observation was that language standardization is a social process. Especially in mature communities, politics, personalities, and historical baggage play a major role in how decisions are made.

There was broad interest in formal tools and mechanization. Whether used to generate test cases or create reference implementations, many projects leaned on semi-formal or fully formal tools to improve precision and reduce ambiguity. However, even the most tool-oriented presentations maintained an awareness of the trade-offs between mathematical elegance and user-driven design, reinforcing the idea that programming languages sit at the intersection of theory and practice.

The workshop also highlighted meaningful divergences in goals, methods, and philosophies. Talks differed in terms of the maturity of the language ecosystems. Projects like Hylo represented fresh attempts to build a language from scratch, giving them freedom to define clean abstractions without legacy constraints. On the other hand, talks on JavaScript, Scheme, APL, C++ dealt with "hardened" ecosystems, where evolution required careful negotiation and respect for historical "quirks".

A notable distinction appeared in the languages position standardization processes. Some focused on formal, international bodies, while others described more organic or community-driven processes. These differences shaped both the speed and structure of design feedback loops.

We present below summaries of the talks given at the Workshop.

2. PLSS 2025 Talks

2.1. Keynote: Engineering a Formal Language Specification by Andreas Rossberg

Summary of the talk The speaker presented an overview of the design, formalization, and mechanization of the WebAssembly specification, with a particular focus on SpecTec (Youn et al. 2024), a domain-specific language and toolchain developed to streamline the specification process.

The presentation began by tracing the roots of formal language specification to the 1990 *Definition of Standard ML*, one of the first programming languages to be fully defined using operational semantics inference rules. Inspired by this, WebAssembly was engineered from its inception with a complete formal semantics as a normative part of its specification.

To ensure the soundness and safety of the language, WebAssembly's semantics have been fully mechanized in theorem provers Coq and Isabelle (Watt et al. 2021). These provide machine-verified guarantees the absence of undefined behavior; this is something which is uncommon in industrial languages.

As WebAssembly has grown in size and complexity, maintaining its multi-representation specification (formal rules, prose descriptions, reference interpreters) became increasingly costly and error-prone. To address this, the SpecTec DSL that allows the formal semantics to be authored once and then used to generate all necessary artifacts has been developed.

The speaker demonstrated the use of SpecTec by defining a mini-language "NanoWasm", showing how abstract syntax, typing rules, and execution semantics can be expressed and automatically transformed into specification documentation. SpecTec enables consistent generation of formal and natural language outputs from a single specification source. SpecTec has been recently adopted by the W3C WebAssembly Community Group as the official authoring tool for the specification[1].

The presentation concluded with a look ahead: potential applications of SpecTec include test generation, verified interpreters, symbolic execution engines, and auto-generated compilers.

The talk emphasized that SpecTec represents a major advancement in how industrial language specifications can be engineered and maintained.

Q&A session

- **Q:** *Do the formal principles applied to WebAssembly also apply to SpecTec itself?* **A:** Not yet. While SpecTec acts as a preprocessor with clean internal semantics, its own meta-theory has not been formally specified yet.
- **Q:** *How many users are expected to use SpecTec?* **A:** It is expected to remain a specialist tool with a small user base (likely a two-digit number). While not highly polished, it already significantly improves over previous tooling. If other projects using SpecTec succeed, broader adoption might push usability improvements.
- **Q:** *Given SpecTec is an external DSL, is there any support like IDE plugins or syntax highlighting?* **A:** Currently, no. Tooling is minimal, and there is no IDE support yet. However, even without that, SpecTec provides a more stable and controlled environment than the prior toolchain. IDE support would be ideal but is a lower priority.
- **Q:** *Can SpecTec express language properties, such as "well-typed programs do not go wrong"?* **A:** Not currently. This is a desired feature under discussion, particularly for theorem prover integration. The challenge is in designing a general enough syntax to express theorems that can map to different theorem provers.
- **Q:** *Were you inspired by CPU specification tools, like Sail[2]?* **A:** Yes, tools like Sail were looked at. However, the requirements differ significantly, and WebAssembly's needs warranted a more specialized solution.
- **Q:** *How does the animation step convert declarative rules to algorithmic prose?* **A:** This is handled by a hardcoded interpretation in SpecTec's backend. It transforms declarative reduction rules into algorithmic prose by pattern matching and identifying the next reducible expression. For WebAssembly, this works well due to its simple, sequential nature, but could be harder for more complex languages.

2.2. APL Array Notation by Adám Brudzewsky

Summary of the talk The speaker summarized the ten-year development of a general-purpose array literal syntax for APL (Brudzewsky 2020), filling a long-standing gap in the language's capabilities.

[1] https://webassembly.org/news/2025-03-27-spectec/
[2] https://github.com/rems-project/sail

The talk began with a historical overview of APL's evolution from mathematical formalism to an interactive programming language used to specify major computing systems like IBM's System/360. The speaker demonstrated APL's implicit and expressive style, where symbols like ρ ("reshape") and $+$ automatically broadcast over arrays, without the need of loops or declarations. While this symbolic design is powerful, it created challenges in writing or reading complex literal arrays, particularly multi-dimensional or nested structures.

Early proposals explored using square brackets with special symbols like $\diamond$ ("diamond") to indicate rows, aiming to enable multi-line JSON-like syntax. However, the design had to carefully avoid ambiguities caused by APL's overloaded use of symbols for indexing, assignment, and expression composition. Through iterative development, the team arrived at a system where square brackets represent tables, round parentheses represent lists, and parentheses also enclose objects, thus achieving both clarity and compatibility.

The notation was implemented in three stages: a prototype in APL, a C++ tool callable from the interpreter, and a native implementation in the Dyalog APL interpreter. The resulting system allows developers to write and read arrays (including deeply nested structures) in a clear, unambiguous form, compatible with APL's interactive and symbolic nature.

While the array notation has not yet been adopted as an official standard, a complete specification and rationale are available publicly. Efforts to coordinate with other APL vendors have met with mixed success, reflecting a broader fragmentation in the ecosystem.

The speaker emphasized that the absence of a formal standardization committee enabled faster iteration, but also left unresolved tensions around interoperability and long-term adoption. The notation is now shipping in Dyalog APL version 20, offering a practical solution to a long-standing problem in the APL world.

Q&A session

- **Q:** *Given the lack of a standard syntax definition in APL, are there plans to create a more formal definition that multiple implementations can share?* **A:** There are no concrete plans at present. As a commercial vendor, Dyalog is hesitant to unilaterally define a standard. Coordination with other vendors has proven difficult. However, a neutral entity called the *APL Trust* is being established, which might lead such an effort in the future. Standardization would be challenging due to divergence among existing implementations.
- **Q:** *It seems like a benefit of your process was avoiding a standardization committee. But the drawback was also the lack of joint design across implementations. Can you reflect on that tension?* **A:** Yes, it is a real tension. The absence of a formal committee allowed the design to evolve much faster. A committee would likely have delayed progress for decades. Still, the resulting lack of cross-vendor design has made adoption and interoperability harder, especially for advanced or modern features.
- **Q:** *Would it make sense to at least standardize the common core that all APL implementations share?* **A:** In theory, yes. In fact, two ISO standards for APL already exist[3], with the most recent covering much of the common core. However, that standard has issues and has not been widely adopted by vendors. A practical appendix or update defining a truly portable core might be viable.
- **Q:** *In the array notation, is there ambiguity between using empty parentheses for an empty list vs. an empty object?* **A:** There is no ambiguity in practice. In APL, all arrays must have a type. An empty literal without a type cannot represent an empty list, because it lacks type information. Therefore, `()` is unambiguously used to represent an empty object. Empty arrays must be constructed explicitly using APL primitives with type-aware emptying.

2.3. Trusted JavaScript Language Environments with ESMeta by Jihyeok Park

Summary of the talk This speaker presented ESMeta (Ryu & Park 2024), a toolchain developed to address the growing complexity of ensuring conformance and correctness in JavaScript implementations. As JavaScript evolves rapidly, with the ECMA-262 specification updated annually by TC39, developers and tool authors face increasing challenges in aligning with the latest semantics. ESMeta offers an automated solution by extracting a mechanized (i.e., executable) specification from the prose of the standard, providing a basis for deeper analysis, verification, and tool development.

The presentation first introduced the motivation: the ECMA-262 specification is lengthy and informal, making manual conformance checking error-prone and unsustainable. To solve this, a meta-language tailored for ECMA-262 has been developed, as well as a tool that parses and translates the prose algorithms into executable code. This enables a range of applications. One is type analysis of the specification itself, which uncovered over 90 specification issues, including 14 previously unknown bugs. Another is automated conformance test synthesis, generating JavaScript programs from the mechanized specification, to test engine behavior. This enabled finding over 40 bugs in real-world engines and over 90 bugs in transpilers.

ESMeta also supports visualization and debugging: a Chrome extension allows specification users to inspect specific algorithms, view minimal example programs that trigger them, and interactively trace their execution using a "double debugger" that steps through both the specification and the program simultaneously.

A further contribution is deriving correct-by-construction tools, such as a static analyzer for JavaScript, by analyzing the interpreter generated from the mechanized specification. Although slower than handwritten tools, this approach offers guaranteed correctness for modern language features and informs future tool development.

The speaker concluded by highlighting the importance of structuring specification prose in a regular style, noting that ESMeta's effectiveness depends on such consistency. ESMeta

[3] ISO/IEC 13751:2001 and ISO 8485:1989.

has been integrated into the continuous integration pipeline of ECMA-262.

Q&A session

- **Q:** *Who are the actual users of the interactive debugger?* **A:** The tool is used by three main groups: ECMA-262 specification editors and maintainers, proposal authors, and developers of JavaScript language tools. It is especially useful for understanding the behavior of edge cases.
- **Q:** *Are specification writers actually using the debugger to debug the specification?* **A:** Yes, though it is called a debugger, it is often used more for understanding and exploring the semantics of JavaScript rather than debugging per se.
- **Q:** *The assertion examples shown were simple (e.g.,* `x = 42`*). How do you generate assertions for complex values like objects or functions?* **A:** The assertion generator supports complex JavaScript features, including prototype chains, property descriptors, and property ordering. However, it does not attempt to assert the behavior of functions, which is significantly more complex.
- **Q:** *Could the intermediate representation be reused for other programming languages to get similar tools automatically?* **A:** Technically yes, but ESMeta's metalanguage is tailored to JavaScript semantics. For other languages, it would be better to design a custom metalanguage that reflects their unique semantic properties.
- **Q:** *The ECMA-262 specification has become more stylized (e.g., question mark/bang notation). How much do you depend on this structure? Would your system work without it?* **A:** ESMeta strictly follows the current style of the specification. The success of the approach depends heavily on that regular structure, but no new style is imposed.
- **Q:** *How often do changes to the specification break your tools?* **A:** Significant updates that require tool changes happen only a few times a year.

2.4. Designing Hylo, a programming language for safe systems programming by Dimi Racordon

Summary of the talk The speaker shared insights from the ongoing development of Hylo (Racordon et al. 2022), a general-purpose programming language designed for safe and efficient systems programming. Hylo is grounded in the principle of mutable value semantics, which promotes programming with values (rather than references) while retaining support for mutation. This model aims to combine the safety and local reasoning of functional programming with the performance benefits of imperative models.

The speaker outlined Hylo's three main goals: safety by default (e.g., eliminating undefined behavior), speed by definition (allowing high-level abstractions without performance trade-offs), and expressiveness through modern language features.

The language traces its origin to an academic effort exploring compiler construction for value semantics. Hylo evolved into a full language design effort, backed by key collaborators including Dave Abrahams. The speaker humorously recalled the first commit message, "*Fight another dragon*", noting that the scope and complexity of the project quickly grew beyond early expectations.

Reflecting on this journey, the speaker shared several lessons. Prospective language designers need to realize that implementing a new language involves far more than writing a compiler; it requires developing an entire ecosystem, including IDE integration, tooling, libraries, and community engagement. The economic and practical barriers to language adoption are substantial. Nevertheless, Hylo has grown an active contributor community, which has helped guide design decisions and maintain developer morale.

The speaker emphasized the value of iterative design: ideas should be implemented and tested early and often, with a tight feedback loop between specification and real-world experience. It is also important to define a clear target audience (in Hylo's case: C++ developers seeking a modern, safer alternative).

Q&A session

- **Q:** *Would you follow your own advice to "not design a language" if you were starting again today?* **A:** No, I would not. I am too passionate about it. Despite the difficulties, I do not regret starting.
- **Q:** *What are your next plans for Hylo?* **A:** We aim to have a preview version of the language ready in about six months. The focus now is to encourage real-world usage and prepare for outreach.
- **Q:** *You mentioned pressure to finish: how do you know when a language project is "done"?* **A:** It never really ends. But you can identify milestones, where you have solved a core problem. At that point, others can take over to iterate and refine the system.
- **Q:** *Building a language is more than a compiler: how do you view the scope of ecosystem expectations?* **A:** I fully agree. A language project today means building tools, libraries, documentation, and more. Without immense resources, it is almost impossible to match expectations. Do it only if you truly enjoy it.
- **Q:** *When should one start building tooling for a new language?* **A:** As soon as you are writing programs that are longer than 4 lines. Tooling improves your own development experience (e.g., debugging the standard library), and helps plan for long-term extensibility.

2.5. Just-in-time Specification: Evolving Kotlin One Feature at a Time by Marat Akhin

Summary of the talk The speaker presented Kotlin's approach to language evolution and specification (Akhin & Belyaev n.d.), shaped by its status as a single-vendor, single-implementation language. Despite lacking external standardization pressures, Kotlin has grown significantly over 14 years, targeting multiple platforms (JVM, JavaScript, WebAssembly, and native) while preserving internal consistency. This evolution is guided by a highly pragmatic process that introduces "just-in-time specifications"—formal descriptions of language behavior developed only when needed.

The presentation outlined the lifecycle of a Kotlin language feature, beginning with either external feature requests (often

problem-driven) or internal ideas (often solution-driven). To proceed, both a real user-facing problem and a viable solution must be present. Feature proposals are captured in Kotlin Evolution and Enhancement Process (KEEP) documents, which include motivation, technical design, use cases, implementation details, risks, and a just-in-time specification fragment. These fragments are prose-based, publicly available (even internal-facing) documents. While this approach reduces cost and increases speed, it requires strong internal discipline and can miss unspecified dependencies, especially in multiplatform contexts.

The speaker then emphasized Kotlin's layered decision-making process: each KEEP must be internally driven by a Kotlin team member to ensure depth, consistency, and maintainability. External suggestions are welcomed but are vetted internally. The design team analyzes GitHub codebases and internal projects to validate whether a problem occurs widely in practice, and rejects many suggestions early if the cost-benefit ratio is unfavorable. In cases when the design process reaches a tie, the lead designer has the final authority, based on the understanding that the team will iterate and refine over time.

A central theme was language feature minimalism: the best features are those the language does not need to add. Kotlin leverages libraries, IDE support, and compiler plugins as alternatives to core language changes. If a feature proceeds, it is tested internally before being released through experimental and pre-stable phases.

Q&A session

- **Q:** *Is Kotlin's language evolution process community-driven or solely led by JetBrains?* **A:** It is driven by the Kotlin team at JetBrains. While community input is considered (especially through feature requests), the KEEP process requires internal maintainers, as the team is ultimately responsible for the language's coherence and success.
- **Q:** *Is there a connection between Kotlin's centralized decision-making and the fact that there is only one compiler implementation?* **A:** Yes. The single-vendor, single-compiler structure allows for cohesive control and simplifies coordination. But it also means that specification responsibilities rest entirely with the Kotlin team.
- **Q:** *Does just-in-time specification put extra burden on feature contributors who must backfill missing specifications?* **A:** Yes, that is correct. It does shift responsibility to contributors. However, most features build on already-specified areas. When gaps arise, the design team either fills them as blockers or uses implementation behavior as a reference. But the trade-off is acknowledged.
- **Q:** *Specifications act as shared documentation and enable onboarding, adoption, and education.* **A:** Agreed. Several users at KotlinConf shared that they use the specification as a way to explain features to colleagues and justify Kotlin adoption.

2.6. The Software Supporting the JavaScript Language Specification by Michael Ficarra

Summary of the talk The speaker gave an in-depth tour of the infrastructure that underpins the development and maintenance of the ECMAScript specification (ECMA-262) (Guo et al. 2025), the official definition of the JavaScript language. As JavaScript continues to evolve in both scale and usage, the need for a robust, maintainable, and accessible specification has grown significantly.

The specification is authored in highly structured English prose, resembling a stylized pseudocode with rigorously defined terminology. This structure balances precision and accessibility across a diverse reader base, including implementers, educators, developers, and researchers. Although still "formally informal", the specification is detailed enough to guide multiple conformant implementations with few divergences.

To support its growth, the editing process shifted post-2015 from a single editor using Microsoft Word to a decentralized system of editor groups and champion-led proposals. Each proposal moves through a multi-stage pipeline, with specification text authored by champions early in the process. This distributed model demanded new tooling to manage complexity, maintain editorial consistency, and support collaborative development.

The speaker outlined the suite of web-based tools used to build and maintain the specification, with web technologies chosen for their openness, extensibility, and ubiquity. The specification webpage also includes interactive features to aid comprehension: hoverable links, alias highlighting, operation pinning, and conditional annotations showing where arbitrary user code may execute. These features are essential for implementers navigating complex semantics.

Despite advances, challenges remain, particularly with producing the required yearly PDF version of the specification. Because no browser supports the W3C's print standard for rich formatting, the committee relies on a proprietary solution. Additionally, accessibility remains an ongoing concern, especially for screen readers handling math-like formatting.

The speaker closed by outlining future opportunities, such as building a semantic diff system for proposals, embedding simulation tools into the specification, formalizing editorial conventions, and developing collaborative design environments.

Q&A session

- **Q:** *Is there real demand for PDFs of the specification?* **A:** Yes, surprisingly. Many implementers prefer PDFs for readability, better typesetting (especially for math), and the ability to print out sections for quick reference.
- **Q:** *How do you handle accessibility for visually impaired readers?* **A:** The team has considered accessibility, e.g., by using high-contrast colors and readable fonts. However, areas like diagrams and math are still problematic. We would welcome help from accessibility experts to improve this further.
- **Q:** *Is the need for interactive tooling a sign that the language or the specification is too complex?* **A:** Possibly. The module system, in particular, is very complex, largely due to legacy decisions made over a decade ago. While it is easy to critique in hindsight, many problems were not foreseeable, and the language has had to evolve to meet changing requirements.

– **Q:** *It is hard to foresee long-term design trade-offs, especially in evolving languages like JavaScript.* **A:** Yes. Iterative development leads to adoption but can also back a language into complexity. Designing everything upfront might be ideal, but rarely practical for languages with JavaScript's scope and pace.

2.7. APL Standards by Karta Kooner

Summary of the talk The speaker discussed how APL's standardization—or lack thereof—impacts the work of language implementers. Drawing from his experience as a developer for Dyalog APL, he provided historical context, technical illustrations, and personal insights into how the APL standards create both structure and friction for developers.

The speaker began with a concise history of APL's evolution; the timeline emphasized that APL development long preceded formal standardization, and continues to outpace it.

The presentation centered around three case studies showing how standards (or their omissions) influence interpreter implementation:

– Parsing ambiguities: due to a lack of operator precedence rules in the standard, identical APL code can be interpreted differently by various implementations. The speaker illustrated this using stranding and indexing examples that behave inconsistently across systems.
– Standard omissions: using matrix inversion as an example, the speaker demonstrated how the standard's refusal to mandate specific algorithms (such as for matrix division) leads to divergent behaviors among implementations when confronted with numerically unstable matrices.
– Intentional divergence: the speaker explained Dyalog APL's choice to diverge from the standard in certain cases, such as comparison tolerance in equality checks and floor operations, arguing that some standard-conforming behaviors would be perceived as buggy or unintuitive by users.

The talk concluded with an appeal to improve future standards: to provide clearer guidance, to reduce ambiguity, and to better support language implementers. The speaker highlighted the broader impact of such gaps on language portability, user expectations, and interpreter correctness.

Q&A session

– **Q:** *So this [function binding] seems to depend on* `successor` *being a function, but that is only something you know dynamically?* **A:** Yes, in APL, the interpreter must resolve names dynamically to determine whether it is dealing with a function or variable, which adds complexity and ambiguity to parsing and evaluation.
Q: *Can you construct a list of functions?* **A:** Yes, you can. Creating an array of functions syntactically works, though it can lead to ambiguity depending on how the interpreter parses the input (e.g., distinguishing between a list of functions and an application of one function to another).
Q: *Are all the differences between APL implementations related to parsing?* **A:** No. While many difficulties stem from ambiguous parsing rules, there are also differences in how primitives behave between implementations, leading to inconsistencies in program behavior.
Q: *Why does matrix inversion sometimes fail on seemingly valid input?* **A:** Because the standard does not specify the algorithm for matrix inversion, implementations behave differently. Dyalog, for example, treats extremely small values as numerical zeros to detect singular matrices, resulting in domain errors that may not appear in other interpreters.
Q: *You mentioned both useful and problematic ambiguities in the APL specification. How do you balance these?* **A:** This is tricky. A practical approach: if unspecified behavior leads to innovation without causing major issues, leave it open. But if divergence would cause problems, or there is clearly only one correct way, then specify it.
Q: *What is the process for evolving the APL specification? Who is involved, and where does it happen?* **A:** Dyalog is increasingly shaping the de facto standard, especially with features like array notation. The newly forming *APL Trust* will likely facilitate cross-vendor dialogue. Informal discussions happen on platforms like `apl.chat`; formal input may come from IBM/APL2, the J community, and others. ISO's role is uncertain due to recent changes and unclear copyright status.

2.8. P4-SpecTec: Mechanized Language Definition for P4 by Jaehyun Lee and Sukyoung Ryu

Summary of the talk The speaker introduced P4-SpecTec[4], a project applying language mechanization to the P4 language for programming packet-processing devices. The talk was structured around three core questions: why mechanize P4, how the mechanization was implemented, and what benefits it brings to the P4 ecosystem.

P4 has four key representations: prose specification, the reference compiler, a test suite, and formalizations. These representations frequently diverge due to independent maintenance and evolving language features, leading to inconsistency. Formal models are often outdated and incomplete, while the prose specification lacks formal precision.

To address this, P4-SpecTec adapts the SpecTec framework (originally built for WebAssembly), to serve as a single source of truth for P4's specification. Drawing inspiration from WebAssembly SpecTec and ESMeta, P4-SpecTec introduces algorithmic inference rules: a middle ground between high-abstraction inference rules and executable pseudo-code. These rules are constructive and executable, enabling the creation of interpreters and type checkers from the spec itself.

This mechanized approach required designing a custom metalanguage and enforcing strict constraints to ensure executability. Using this method, most of the P4 type system has been mechanised. Test generation via guided mutation and fuzzing to trigger specific typing errors has enabled identifying 11 previously unknown bugs in the P4 compiler.

[4] https://p4.org/wp-content/uploads/2024/10/9-2024-P4-Workshop-P4-SpecTec-Mechanized-Language-Definition-for-P4.pdf

In conclusion, P4-SpecTec demonstrates how a robust, mechanized specification model can promote consistency, correctness, and test coverage in evolving language ecosystems. The speaker advocated for the systematic adoption of such techniques for real-world languages.

Q&A session

- **Q:** *You are turning SpecTec into something like a theorem prover: it shouts if things are not marked as input/output. Have you considered using a proof assistant like Isabelle or Coq for this?* **A:** Theorem proving is not the current goal of P4-SpecTec. While interesting, the focus remains on specification mechanization and not formal proof development.
- **Q:** *This feels more like relational programming: you are using constraints to make relations executable. It is closer to logic programming than proof languages.* **A:** I agree. There is similarity to logic programming with mode annotations and discussed potential overlaps with work in WebAssembly SpecTec, including transformation passes to infer mode information.
- **Q:** *Sometimes declarative rules (like subtyping transitivity) are preferable for proof clarity, but making them algorithmic is ugly.* **A:** Agreed. Declarative forms are more elegant for reasoning, while algorithmic forms are necessary for execution. Both have trade-offs, and it is valuable to support both styles where possible.

2.9. C++ Standardization: Reflections and Lessons Learned by Jaakko Järvi

Summary of the talk In this personal reflection, the speaker offered insights into how the language is specified, evolved, and shaped by a vast and decentralized international community. The talk explored both technical and social dimensions of standardization, with illustrative anecdotes from the speaker's own work on major C++ features (such as lambdas and variadic templates).

C++ standardization is a volunteer-driven, paper-based process, governed by ISO rules where each country has one vote. Committees meet thrice annually, with multiple specialized working groups handling areas like core language, libraries, and language evolution. While open to all, effective participation often hinges on persistent advocacy and technical expertise. Final decisions are reached by consensus, but informal dynamics, personalities, and political considerations often play a significant role.

The speaker illustrated how large-scale change happens slowly, using the introduction of lambda expressions as a case study. The lambda feature emerged from years of preparatory work in related areas. The speaker emphasized how incremental refinement and inter-feature alignment enabled the eventual inclusion of lambdas with minimal disruption.

In conclusion, the speaker advocated for incrementalism and transparency in evolving complex languages like C++. Successful standardization balances bold innovation with cautious, consensus-driven progress.

Q&A session

- **Q:** *How does the one-country-one-vote system work when large international corporations employ contributors across many countries?* **A:** Each contributor must be formally registered through their country's national standards body, regardless of employer. ISO requires affiliation through a national entity, not citizenship. The final vote is cast by each country's official delegate.
- **Q:** *Why does ISO use this one-vote-per-country model? Is it to reduce corporate influence?* **A:** Likely so. The intent is to avoid single-company dominance and align with ISO's general approach for technical standards across domains (e.g., electrical plug standards). The structure promotes national rather than corporate influence.
- **Q:** *W3C requires membership fees and restricts voting rights to formal members. In our process, anyone can participate and vote. Are there concerns that this can be abused?* **A:** Yes, similar concerns exist in ISO too. While anyone can attend and informally influence direction, official votes are restricted to national delegations. Generally, the process functions smoothly due to social norms and community oversight.
- **Q:** *Can single individuals derail the process by blocking or manipulating decisions?* **A:** Yes, especially those with long-standing authority or influence. Informal control via framing, timing of straw polls, and selective confusion can sway group opinion. However, such behavior is rare and mostly the process is collaborative.

2.10. Do Programming Languages Fulfill Requirements? Should They? by Michael Sperber

Summary of the talk In this reflective closing talk, the speaker examined the often uneasy relationship between requirements engineering and programming language design, drawing on his experience editing the Scheme R6RS standard and analyzing the evolution of Emacs Lisp. He questioned whether and how programming languages should follow the same requirement-driven processes used in software engineering and proposed a nuanced framework balancing principles, practicality, and long-term evolution.

The speaker argued that programming languages are themselves software artifacts and should be compared to regular software systems. He contrasted the lengthy, deliberative process of the R6RS Scheme standard with modern agile practices, which emphasize short feedback loops and rapid iteration. Unlike most programming languages, Scheme's standard included a set of principles resembling requirements (e.g., support for readability, portability, performance, educational utility); these helped focus the process, though they were often vague or contradictory in practice.

The speaker highlighted that designing for long-term use requires deep attention to modularity, evolvability, and minimizing coupling, echoing software architecture principles from the 1970s.

The speaker then addressed the tension between mathematically principled language design and requirements-driven pragmatism. While languages like Scheme historically drew from

elegant formal systems, many of these principles later proved practically beneficial, often because they align with how humans think and recognize patterns. At the same time, the speaker warned against blindly following either requirements or theoretical elegance, advocating instead for continuous user feedback and incremental evolution, especially in mature ecosystems.

To conclude, the speaker recommended that language designers explicitly state their goals and trade-offs, combine principled foundations with practical needs, and remain open to revising both in light of evolving user experience.

Q&A session

- **Q:** *Why do structures like sums, products, or lambdas fit so well into programming? Are they "natural" for our brains, or is it due to our education?* **A:** It is not about innate brain wiring versus schooling, it is that mathematics identifies patterns our brains are good at recognizing. These constructs are not physically embedded in the universe, but in our understanding of it. They help organize software domains effectively because they resonate with how we mentally structure problems.
- **Q:** *These constructs (e.g., sums, lambdas) are canonical because they mirror logic (e.g., Curry-Howard). Their usefulness is not surprising—they are the minimal building blocks.* **A:** Agreed in part, though I lean toward seeing them as human-centered abstractions rather than eternal truths. They work because they help us understand the world, not because they are woven into it.
- **Q:** *The elegance and internal consistency of principled models is satisfying, but it contrasts with requirements, which are piecemeal and reactive.* **A:** Very true. Requirements can be messy, and principled models offer clean abstractions. But language designers must engage both forces and use each where it serves best.
- **Q:** *Many languages do not plan for removing features. Should we design for graceful feature deprecation?* **A:** Ideally, language evolution should not require removing features, as removal breaks programs. But if removal is truly needed, that often signals the need for a new language. Alternatively, systems like Racket or Emacs Lisp allow file-level language variants, which can enable controlled divergence.

Acknowledgments

The organizers are thankful to the Department of Informatics at the University of Bergen for the financial support of the workshop.

References

Akhin, M., & Belyaev, M. (n.d.). *Kotlin language specification.* https://kotlinlang.org/spec/pdf/kotlin-spec.pdf. (Accessed: 2025-07-05)

Brudzewsky, A. (2020). A notation for APL arrays. *APL-Journal*, *2020*(1–2). Retrieved from https://apl-germany.de/wp-content/uploads/2021/11/APL_Journal_2020_1u2.pdf

Guo, S., Ficarra, M., & Gibbons, K. (2025, June). *ECMAScript 2025 language specification* (ECMA-262 No. 16th Edition). Ecma International. (https://ecma-international.org/publications-and-standards/standards/ecma-262/)

Racordon, D., Shabalin, D., Zheng, D., Abrahams, D., & Saeta, B. (2022). Implementation strategies for mutable value semantics. *Journal of Object Technology*, *21*(2), 2:1-11. Retrieved from http://www.jot.fm/contents/issue_2022_02/article2.html (ECOOP 2021 Workshops)

Ryu, S., & Park, J. (2024, May). Javascript language design and implementation in tandem. *Commun. ACM*, *67*(5), 86–95. Retrieved from https://doi.org/10.1145/3624723

Watt, C., Rao, X., Pichon-Pharabod, J., Bodin, M., & Gardner, P. (2021). Two mechanisations of WebAssembly 1.0. In M. Huisman, C. S. Pasareanu, & N. Zhan (Eds.), *Formal methods - 24th international symposium, FM 2021, virtual event, november 20-26, 2021, proceedings* (Vol. 13047, pp. 61–79). Springer. Retrieved from https://doi.org/10.1007/978-3-030-90870-6_4 doi: 10.1007/978-3-030-90870-6_4

Youn, D., Shin, W., Lee, J., Ryu, S., Breitner, J., Gardner, P., … Rossberg, A. (2024, June). Bringing the WebAssembly standard up to speed with SpecTec. *Proc. ACM Program. Lang.*, *8*(PLDI). Retrieved from https://doi.org/10.1145/3656440

About the authors

Mikhail Barash is an Associate Professor of Programming Languages at Bergen Language Design Laboratory at the University of Bergen, and a co-convener of Task Group on "Experiments in Programming Language Standardization" of Ecma International Technical Committee 39. His interests include Language Specification, Language Workbenches, and Design of Widely Adopted Programming Languages. You can contact the author at mikhail.barash@uib.no.

Yulia Startsev is a Senior Staff Engineer at Mozilla, where she is working on the development of the SpiderMonkey JavaScript engine. She is a co-convenor of Task Group on "Experiments in Programming Language Standardization" of the Ecma International Technical Committee 39, and a former co-chair of the committee. Her main focus is Design and Implementation of Widely Adopted Programming Languages and the Web Platform. You can contact the author at yulia@mozilla.com.