Admissibility All the Way Down

Functional Programming and the Geometry of Reachable Programs

Flyxion — Working Essay

An interpretation of Simon Payton Jones, "Functional Programming, Thinking in Types, Useless Languages" (2025 interview)

> *When the limestone of imperative programming has worn away, the granite of functional programming will be revealed underneath.*
>
> Simon Payton Jones

> *Reality is Reachability. Keep the branches open.*
>
> Flyxion, *The Admissibility Program*

1. Framing the Encounter

Simon Payton Jones has spent over four decades making one bet: that programming with values rather than mutation reveals something more fundamental about computation than the accident of how our hardware happened to be built. In a long 2025 interview covering functional programming, type systems, the future of Haskell, and the implications of large language models, he offers a set of claims that are, from the perspective of the Admissibility Program, recognisable in unexpected depth.

This essay is not a review of the interview. It is an attempt to read Payton Jones's account through the conceptual vocabulary of the Admissibility Program — the research framework developed under the name Flyxion — in order to see what each illuminates in the other. The central claim I will develop is this: functional programming, as Payton Jones describes it, is the engineering discipline of explicitly managing admissibility structure. Type systems are admissibility geometries.

Monads are scope-bounding mechanisms in the sense of Spherepop. The useless-to-useful trajectory of Haskell is a worked example of what it costs and what it reveals to take admissibility constraints seriously over long timescales.

2. Two Computational Primitives and Their Ontological Commitments

Payton Jones opens with a contrast that is, at its core, a metaphysical one: Turing machines proceed by mutation of a tape; lambda calculus proceeds by reduction without any notion of a mutable cell. He notes — emphasising that this is far from obvious — that both models compute exactly the same functions. They are computationally equivalent but ontologically very different.

It is worth being precise about what kind of connection to draw between this contrast and the Admissibility Program. The correspondence is heuristic and structural, not formal or equivalential. The claim is not that Turing machines *are* RSVP or that lambda calculus *is* Spherepop. The claim is that the two computational models differ along the same axis that separates RSVP from Spherepop, and that recognising this illuminates what is at stake in both cases.

RSVP is fundamentally field-theoretic: it describes processes through continuous variation of scalar concentration, vector flow, and entropy density over admissibility landscapes. Its primitives are gradients, flows, and relaxations — things that evolve continuously over a globally accessible substrate. Spherepop is fundamentally event-theoretic: its primitives are discrete, irreversible commitments — formation, collapse, refusal, binding, residue. Each event eliminates possibilities and leaves a residue that constrains what follows.

Along these dimensions, the analogy runs as follows. A Turing machine maintains a globally accessible mutable tape; its computation is evolution over a shared mutable substrate, tracking a continuously updated state via a program counter. This resembles field dynamics: the tape is the substrate, the head position and register contents are the field values, and computation is the propagation of updates across that field. Lambda calculus, by contrast, has no global state. Computation is constituted entirely by reduction events: each beta-reduction irreversibly eliminates a redex, commits to a normal form, and leaves a residue (the substituted body) that constrains subsequent reductions. There is no persistent substrate —

only an accumulating history of committed reductions.

The analogy should not be pressed to formal equivalence. A Turing machine can be given an event-theoretic presentation; a lambda term can be given a field-theoretic denotational semantics. What the heuristic correspondence captures is the *default orientation* of each computational primitive: Turing's model makes global mutation its primitive operation and history a derived record; Church's model makes irreversible commitment its primitive operation and global state a derived convenience.

Payton Jones's lifelong wager — that programming with values reveals something more fundamental than programming with mutation — is therefore, in Admissibility Program terms, a claim about which orientation is primary. His hypothesis is that the event-theoretic orientation (reduction, commitment, irreversibility) is the bedrock, and that the mutation-based orientation (mutable state, program counter, register modification) is a derived convenience built on top of it. Whether or not the RSVP-Spherepop framework vindicates that hypothesis in full generality, it provides one formal context in which the hypothesis can be stated with precision.

3. Side Effects as Admissibility Violations

The core technical achievement of Haskell, as Payton Jones describes it, is to make side effects visible in the type system. A function of type `Int -> Int` cannot perform IO. A function of type `Int -> IO Int` can. This is not merely a stylistic convention; it is a structural constraint on the set of admissible programs.

In the vocabulary of the Admissibility Program, a type signature is an admissibility specification. It defines the space of reachable futures for a given computation: what inputs are acceptable, what outputs are possible, and — crucially in Haskell's case — what effects are permitted along the way. A type- incorrect program is not merely a wrong program; it is an inadmissible program, one that has exited the set of reachable states the system is designed to navigate.

Payton Jones identifies two consequences of this. The first is safety: 99% of the security exploits that arise from buffer overruns and pointer manipulation would be eliminated by construction in a sufficiently typed language, because those exploits are inadmissible states that a properly specified type system simply

refuses to reach. The second is maintainability: after 35 years of development on GHC, Payton Jones still makes large-scale systematic refactorings "fearlessly" because the type system keeps him safe. He changes a few type signatures and then follows a wave of type errors through the codebase, each error pointing to a place where the system's admissibility structure has been violated and must be restored.

This is Repair Theory in action, without the name. The type system is the repair infrastructure of the codebase. Types define what an admissible state of the program looks like. Type errors identify sites where admissibility has been lost — where the program has entered an inadmissible configuration. The programmer's task is to restore admissibility, which the compiler actively assists by identifying every site where the repair is needed. This is maintenance of admissible states, not construction from scratch.

Payton Jones's image of the 10-year-old codebase written by a programmer who left long ago, with invisible coupling through shared mutable global variables, is precisely what the Admissibility Program calls *extraction*: the projection has been mistaken for the territory, the hidden coupling has been made invisible, and the result is that the actual admissibility constraints of the system are no longer legible. The type system, by contrast, forces admissibility structure into the open. The invisible coupling between functions through shared mutable state becomes visible as a type dependency; if you change the type, every site of the coupling lights up as an error.

4. Scope as Geometry: Monads and Spherepop

One of the most technically interesting moments in the interview is Payton Jones's account of why Haskell needed monads and what they actually do. The problem is this: functional programming eliminates side effects by default, but real programs need IO. The question is how to admit a controlled class of effects — a bounded region of admissible interaction with the world — while maintaining the overall purity guarantees that make functional programs compositionally safe.

The solution is the IO monad, which Payton Jones characterises as a type-level marker of dirtiness. A value of type `IO Int` is a computation that, when run, will do some IO and return an integer. The type system enforces that these dirty

computations cannot escape into pure code without explicit acknowledgment. The do-notation sequences dirty computations in a specified order, which is how Haskell achieves the controlled execution of effects.

This is a direct implementation of what Spherepop calls *scope as geometry*. In Spherepop, a bubble defines a topological boundary: entering the bubble makes certain futures accessible that are not accessible from outside it, and closing the bubble collapses those possibilities into a residue. The IO monad creates exactly this structure at the type level. Code inside a do-block is inside the IO bubble: it can perform effects. Code outside is outside the bubble: it cannot. The monad's bind operation (`>>=`) is the mechanism by which commitments made inside the bubble are threaded through subsequent computations while keeping their effectful character visible.

The effect system that Payton Jones goes on to describe — where a type can specify not merely *whether* a computation is effectful but *which particular effects* it can perform — is a refinement of this geometry: not just "inside or outside the IO bubble" but a more granular topology of admissible effectful trajectories. The library Bluefin that he mentions, and the effect system development now happening in both Haskell and OCaml, are precisely attempts to specify the shape of the admissibility region more precisely.

There is also a connection to what Spherepop calls *refusal*. The `unsafePerformIO` escape hatch — the function that lets you perform IO while telling the type system to pretend you haven't — is named with explicit acknowledgment of its danger. The name is itself a documentation of a boundary- crossing: the programmer is asserting that they are taking on responsibility for an action that the formal admissibility structure of the type system would refuse to sanction. This is not quite a Spherepop refusal event in the technical sense, but it shares the key characteristic: it is an autonomous act of stepping outside the normal scope constraints while explicitly accepting the consequences, rather than a silent violation of them.

A further connection worth drawing, which the interview itself only approaches obliquely, is to the Admissibility Program's concept of Markov boundaries. Pure code and effectful code in Haskell are separated by an informational boundary: information can pass from pure code into effectful code freely (you can use a pure value anywhere inside a do-block), but effectful computation cannot flow back into pure code without explicit sanctioning through the type system. The IO monad is, in this sense, a computational Markov boundary: it separates two causal domains

(pure and effectful) and specifies the sufficient statistics — the type signature — that are needed to communicate across the boundary. The monad is not merely a container for dirty computations; it is a boundary that makes the causal structure of the program legible at the type level. The effect systems now being developed in both Haskell and OCaml are refinements of this boundary: instead of a binary pure/impure distinction, they describe a more granular topology of causal domains and the specific interfaces through which information can cross between them.

5. Laziness, Admissibility, and Reachable Futures

Payton Jones's account of lazy evaluation offers another point of contact. In a lazy language, expressions are not evaluated until their values are actually needed. The standard example he gives is a chess program that generates an infinite tree of possible moves: in a strict language, the tree must be generated before it can be explored, which is impossible; in a lazy language, the generator and the explorer can be modularly separated, with the generator producing only as much tree as the explorer actually consumes.

This is a computational implementation of the Admissibility Program's core ontological claim: reachability is more fundamental than location. In a strict language, all of the tree is brought into existence (evaluated, located) before exploration begins. In a lazy language, only the reachable parts of the tree — the parts that the explorer's trajectory actually visits — are ever instantiated. The lazy language implements a computation in which admissibility structure determines what exists, rather than the converse.

Payton Jones's phrase "lazy evaluation is very powerful glue that lets you glue together two programs that you'd like to be distinct" is, in Admissibility Program terms, a claim about how compositional modular design preserves the admissibility structure of each component. Strict evaluation "forces you to merge them together" — it collapses the distinction between the generator and the explorer, destroying the modularity. This is extraction in CLIO's sense: the projection of the combined system discards the structural independence of its components.

6. Type Systems as Admissibility Geometries

Payton Jones's technical account of type systems contains, in compressed form, a theory of what it means to specify an admissibility geometry. Let me draw out the key structural claims.

A type system is a mechanism for specifying, at compile time, which programs are admissible. A dynamically typed language accepts all syntactically well-formed programs as admissible; whether a program actually terminates or produces a sensible value is discovered only at runtime. A statically typed language restricts the admissible programs to those that satisfy the type constraints, eliminating at compile time a large class of programs that would fail at runtime.

The tension Payton Jones identifies — between type systems that are too weak (rejecting programs they should admit) and type systems that are too strong (hard to use) — is precisely the tension between an admissibility region that is too narrow and one that is too broad. Too narrow: you exclude well-behaved programs like "apply list-reverse to a list of characters" because your type language is not expressive enough to specify that the operation is admissible regardless of element type. Too broad: you admit programs that fail at runtime because your admissibility specifications are insufficiently discriminating.

The goal is to find an admissibility geometry that is both maximally inclusive of programs you want to run and maximally exclusive of programs you do not. Parametric polymorphism (for all α, list of α to list of α) is a move that expands the admissibility region in exactly the right direction: it admits more programs (reverse works on lists of any type) without admitting any bad ones (you still cannot confuse integers with floating point numbers). Type classes and effect systems are further moves that refine the geometry.

This maps directly onto what the Admissibility Program calls the *projection-invariance problem* in CLIO: what features of an admissibility landscape survive projection into a lower-dimensional representation, and how can they be distinguished from artefacts of the projection operator? A type system is precisely a projection of the full semantic landscape of a program into a more tractable representation. The question of which type system to use is the question of which projection preserves the most relevant admissibility structure without introducing artefacts. Parametric polymorphism is an admissibility-preserving projection that avoids the type-error of Pascal's monomorphic reverse function.

6.1. GHC Core: Admissibility Maintained Through Every Transformation

The most striking formulation in the interview is Payton Jones's account of GHC's internal language Core, which is a statically typed intermediate representation of Haskell programs. The architecture is worth examining in full because it is, without being described in these terms, a nearly complete implementation of the Admissibility Program's concept of repair infrastructure in a compiler.

Core is based on System F, the statically typed lambda calculus defined by Girard and Reynolds. Every Haskell program, after parsing and type-checking, is translated into Core. The key property is this: every optimisation pass in GHC must transform a type-correct Core program into another type-correct Core program. After each pass, GHC runs a type-checker on Core to verify that the optimisation has preserved type correctness. If it has not — if the pass has introduced a type error — that is a bug in the optimiser, caught immediately.

The consequences of this architecture are significant. An optimisation pass that introduces a type error in Core has exited the admissible region of compiler transformations. The type-checker for Core does not verify correctness of the output — it cannot, in general, prove that the optimised program behaves identically to the input. What it does is verify that the output remains *within the admissible region*: that it is still a well-typed program in System F. This is exactly the distinction between verification (proving correctness) and admissibility maintenance (ensuring the system has not left the permitted region of states).

The alternative, which Payton Jones notes is universal among other production compilers, is to have no type checker for the intermediate language at all. In that architecture, a bug in an optimisation pass produces machine code that may crash at runtime — often at a point far removed from the site of the original error. The debugging process requires backtracking from the runtime failure, through the generated machine code, through the intermediate representations, to the specific optimisation pass that introduced the error. This is the structure of what the Admissibility Program calls *extraction*: the admissibility constraints of the transformation have been made invisible, and when they are violated the error propagates irreversibly into the output before it can be detected.

With Core's type-checker, violations are detected locally and immediately. Each optimisation pass is bracketed by an admissibility check. The error is caught at the boundary of the pass that introduced it, not at the downstream consequence. This

is repair infrastructure in the precise sense: the type-checker is a mechanism that continuously monitors whether the system has remained in the admissible region, and flags violations at the earliest possible point so they can be addressed before they propagate.

Payton Jones notes that no other production compiler has this property and expresses genuine pride in it. From the Admissibility Program's perspective, the rarity of this approach is itself diagnostic. We have built an enormous amount of software infrastructure without the repair infrastructure that would allow us to detect admissibility violations before they propagate to irreversible damage. The internet's insecurity — which Payton Jones separately attributes to the absence of type safety in infrastructure languages — and the absence of type-checked intermediate representations in compilers are two faces of the same civilisational failure: systems built without continuous admissibility monitoring, maintained at enormous cost because violations are detected only after they have become irreversible.

7. The Useless-to-Useful Trajectory as Stratum Transition

Payton Jones presents a two-dimensional design space with safety on one axis and usefulness on the other. The first version of Haskell occupied a position of high safety and low usefulness: a Haskell program was simply a function from string to string, with no IO at all. C occupies a position of high usefulness and low safety. The trajectory of Haskell's development has been to move vertically — gaining usefulness without sacrificing safety — while Rust's trajectory has been to move horizontally from C's position, gaining safety while retaining usefulness.

This is a map of an admissibility landscape, though Payton Jones does not use that language. Safety corresponds to the narrowness of the admissibility region — to how precisely the type system specifies which programs can be written. Usefulness corresponds to whether the admissible programs include enough of the practically useful ones. The nirvana quadrant — safe and useful — is the region of the admissibility landscape where the specification is simultaneously tight enough to exclude the bad programs and generous enough to include the good ones.

The history of Haskell is the history of expanding the admissibility region

in the direction of usefulness without relaxing it in the direction of safety. Each extension — monads for IO, type classes for polymorphism, GADTs for dependent-like typing, the effect system now being developed — is a move that admits more programs as safe without admitting more unsafe ones. The constraint is always that the move must not exit the region of formal admissibility guarantees.

In Admissibility Program terms, this is a sequence of *stratum transitions*: each extension corresponds to recognising that the current admissibility geometry was too narrow (it excluded programs that are in fact safe) and expanding it in a way that preserves the constraints that generate the safety guarantees. The development of the IO monad was a stratum transition: it expanded the class of admissible Haskell programs from functions over strings to programs with controlled effects, without relaxing the guarantee that effects outside the IO type cannot contaminate pure code.

8. LLMs, Admissibility Filtering, and Active Geodesic Inference

One of the most interesting claims in the interview is Payton Jones's assertion that static type systems are "a huge boon for LLMs." His reasoning is precise: an LLM generating code in a statically typed language can use the type checker as a filter. It generates a candidate program, runs the compiler, and if the program is type-incorrect it tries again. The type system dramatically tightens the feedback cycle by rejecting inadmissible programs immediately, before they reach any test suite. In an untyped language, the only feedback is from running the program, which may require an elaborate test environment to detect failures.

This is the Admissibility Program's framework of *semantic navigation* applied to code generation. The LLM is navigating a semantic manifold — the space of possible programs — under admissibility constraints imposed by the type system. The type checker is an admissibility filter: it immediately identifies when a generated program has exited the admissible region and provides feedback about where the boundary was crossed (a type error with a location). The type errors are not obstacles to be circumvented; they are signals that guide the navigation back into the admissible region.

This also connects directly to the framework of Active Geodesic Inference. Intelligence is defined in that framework as the capacity to remain within a family

of dynamically admissible, low-action histories by actively reshaping the geometry of configuration space. An LLM navigating the space of Haskell programs with a type checker as an admissibility oracle is implementing something close to this: at each step it generates a candidate trajectory (a candidate program) and receives a signal about whether that trajectory is admissible. The type system provides the geodesic structure — the metric on the program space that determines what counts as a nearby admissible program.

Payton Jones's insight is that this loop closes much faster with a static type system than with runtime testing. The admissibility filter is applied earlier in the pipeline, eliminating large classes of inadmissible trajectories before they are even committed to. This is why semantic isomers — the AGI framework's concept of distinct internal reasoning trajectories that project to the same external observable — matter here: two programs that produce the same output on the test suite may have very different type structures, and the type-correct one is more likely to remain admissible under subsequent modifications.

9. Avoid Success at All Costs: Admissibility Under Historical Pressure

The phrase "avoid success at all costs" that Payton Jones discusses is not merely a witty aphorism. It describes a genuine tension in the management of an admissibility region over time.

Success — in the sense of widespread adoption — creates historical pressure to relax admissibility constraints. Users want to write programs that the current type system rejects; the path of least resistance is to expand the admissible region to include those programs. But if the expansion is not carefully controlled, it admits programs that are unsafe, and the safety guarantees that made the language valuable in the first place are eroded.

The Haskell community's commitment to principled constraint maintenance — its refusal to simply add features because users want them, its insistence on finding the right abstraction rather than the convenient hack — is an instance of what the Admissibility Program calls *repair theory*: the active maintenance of an admissible state in the face of constant pressure toward inadmissibility. The type system is the thing being repaired. The users who want unrestricted side effects are the

source of constraint erosion. The research program of finding new type-theoretic abstractions (monads, effect systems, type classes) is the repair mechanism: it finds ways to extend the admissible region without relaxing the constraints that generate the safety guarantees.

Payton Jones's backward compatibility project — ensuring that a program that compiles with GHC 10.0 also compiles with GHC 10.2 — is a different kind of repair: preserving the admissibility of existing programs as the language evolves. Every language extension is a potential admissibility violation for programs that depend on the old behaviour. Maintaining backward compatibility is the work of ensuring that the admissibility region grows only upward — admitting new programs — without contracting downward, inadmissible programs that used to be admissible.

This is *civilisational repair* applied to programming language design. The codebase of the world's software is the institution; the type system is the governance mechanism; backward compatibility is the undo stack. The decision to invest serious effort in backward compatibility is a decision to treat the existing codebase as infrastructure worth preserving — to apply Repair Theory at the level of an entire programming ecosystem rather than a single codebase.

10. Types as Coordination Geometry

One of the deepest themes in the interview is not correctness but maintainability, and what Payton Jones says about maintainability points toward a dimension of type systems that their usual framing as *correctness tools* does not capture. When he describes doing large-scale refactorings of GHC "fearlessly" because the type system keeps him safe, or when he contrasts the difficulty of maintaining 15-year-old dynamically typed code with the relative ease of maintaining 35-year-old statically typed code, he is not primarily describing a correctness achievement. He is describing a coordination achievement.

A type signature is a contract. When Payton Jones writes a function with type signature `Map k v -> k -> Maybe v`, he is not merely describing the function's behaviour to the compiler. He is making a commitment to every future programmer who will read, use, or modify that code — whether that programmer is a colleague working simultaneously in a different office, a junior developer who will join

the project in five years, or Payton Jones himself in 2040. The type signature communicates across time and organisational distance: it specifies exactly what the function accepts, what it returns, and — through the absence of any IO marker — that it has no side effects on any shared state.

This is what the Admissibility Program calls *coordination geometry*: the structure by which distributed agents can synchronise their local constraints without direct communication. A type system is a propagating constraint field. When a type changes, the type-checker identifies every site in the codebase where that change requires a corresponding adjustment — not through runtime discovery or test failure, but through immediate structural notification. The type error is a coordination signal: it tells every contributor, at every level of the stack, exactly where their local assumptions depend on the changed constraint and exactly where repair is required.

Payton Jones's image of "a wave of changes propagating through" the codebase after a type change is extraordinarily close to how Coordination Geometry describes constraint propagation across a network of coupled agents. The type system is the medium through which this propagation occurs. It does not merely prevent bugs; it enables a form of coordination across time that would otherwise require explicit communication between every pair of programmers who might ever touch related code. In a dynamically typed codebase, that coordination is achieved instead through documentation, convention, testing, and institutional memory — all of which are fragile, invisible, and lossy. The type system makes the constraint structure of the coordination explicit and formal.

This interpretation suggests a reframing of Payton Jones's most provocative claim — that static type systems are "a huge boon for LLMs." The boon is not merely that LLMs receive faster feedback on type errors. It is that type systems provide a formal coordination interface between human programmers and LLM-generated code that does not exist in dynamically typed languages. An LLM generating code with a type signature is making a legible commitment: anyone who reads the generated code can verify, without running it, whether it connects correctly to the surrounding code's constraint structure. The type system is the medium through which human programmers can coordinate with an LLM tool across the boundary of their respective competencies.

Viewed at civilisational scale: the reason functional programming ideas — lambdas, polymorphism, monads, garbage collection — have steadily infected

mainstream imperative languages is not that they are theoretically elegant. It is that they reduce coordination costs. Each of those ideas makes some aspect of a program's constraint structure more visible and more formal, allowing programmers to coordinate more effectively across time, distance, and organisational boundaries. The "limestone" of imperative programming wears away not because functional programming is aesthetically superior, but because the constraint-visibility offered by functional ideas reduces the friction of civilisational-scale software maintenance.

11. When the Formalism Becomes the Interface

A recurring theme in the interview, stated most clearly in Payton Jones's account of how he designs programs, is that experienced Haskell programmers stop using types as annotations and start thinking through types. He writes the types first. He changes a type and follows the wave of errors. He describes the type system as his "design language." This is not a description of a tool for checking designs already conceived; it is a description of a medium in which design itself happens.

This is the thesis of the Admissibility Program's essay *When the Formalism Becomes the Interface*: there is a phase transition in the relationship between a formal system and its user, after which the formalism ceases to be a representation of thought and becomes the medium through which thought occurs. The formalism does not describe the interface; it *is* the interface.

For Haskell programmers at that phase transition, the admissibility geometry of the type system is not an external constraint imposed on their thinking. It is the internal structure through which their thinking is organised. A type error is not "my program failed a check"; it is "I have not yet found the right decomposition of this problem into admissible pieces." The type system becomes an active collaborator in the process of design — something that Payton Jones gestures at when he says he finds that "changing the representation of a data structure" and then compiling tells him exactly where in the codebase that representation is used, written, and freshly allocated.

Null Convention Logic, which Payton Jones cites in the context of hardware for functional programming, is another instance of this phenomenon. In NCL, a circuit is complete — its computation is abstracted — at the moment of signal stabilisation: the formalism of completion-governed logic becomes the medium

through which the circuit expresses its computational boundary. The Admissibility Program's *Abstraction as Reduction* monograph identifies NCL stabilisation as one of five formally distinct instances of the same invariant: inner complexity is resolved to allow outer complexity to grow without conflict. When Haskell's type system becomes the programmer's design language, that same invariant is at work: the inner complexity of type resolution is completed by the compiler so that the outer complexity of system design can proceed without interference.

The phase transition that makes a formalism into an interface is not available to all formalisms. It requires that the formalism be expressive enough to capture the relevant admissibility structure, and complete enough that working within it feels like thinking rather than translating. Payton Jones's assessment that Haskell embodies the single core principle of "no unrestricted side effects" and declines to compromise it is, from this perspective, a prerequisite for the phase transition to occur: a formalism that makes exceptions for convenience cannot become the medium of thought, because thinking through it would require constantly tracking which parts of the formalism apply and which have been suspended.

12. What Remains Unreachable: The Boundaries of the Admissibility Program's Reach

Payton Jones's account also contains points of genuine tension with the Admissibility Program's framework, worth naming explicitly.

The first is performance. He acknowledges that lazy evaluation, despite its theoretical elegance, creates practical problems for performance prediction and memory management. The admissibility constraints that make Haskell's programs compositionally safe also make their performance behaviour less predictable, because the order of evaluation is determined not by the program's structure but by what the consumer actually demands. This is a case where the admissibility geometry of the type system is genuinely constraining in a way that is not straightforwardly captured by the Admissibility Program's framework, which tends to treat constraints as revealing structure rather than hiding it.

The second is the question of what the type system cannot specify. Payton Jones acknowledges that no type system can prevent a program that deliberately exfiltrates its entire database in response to a request. Deadlock, high-level logic

errors, and semantic bugs are outside the reach of any type-based admissibility specification. The Admissibility Program's notion of admissibility is also formal — it specifies what transitions are allowed from a given state — but the relationship between formal admissibility and actual correctness or safety is not straightforwardly compositional. You can have a formally admissible trajectory through a semantic space that leads somewhere genuinely terrible.

The third is the gap between the theory and the ecosystem. Payton Jones is frank that Haskell is talked about far more than it is used. The principled constraint maintenance that makes it theoretically attractive makes it practically difficult, and the community that finds it rewarding is self-selected for people who want to think in a different way. The Admissibility Program is similarly a framework that requires a substantial investment in new ontological vocabulary before it becomes productive. Neither has solved the problem of making the insight accessible without diluting it.

13. The Limestone and the Granite

Payton Jones's most evocative image — "when the limestone of imperative programming has worn away, the granite of functional programming will be revealed underneath" — is a geological metaphor that the Admissibility Program has independently arrived at. The five-layer stratigraphy of the Admissibility Program places practice and embodied knowledge at the bedrock, core intuitions as subsoil, primary theories as mantle, formalization programs as crust, and domain applications as surface. The claim is that the deeper layers are more durable, more structurally fundamental, and more resistant to erosion than the surface features.

Payton Jones's geological intuition is the same: imperative programming is limestone — useful, widely distributed, built from the accumulated sediment of decades of engineering practice, but ultimately soluble. Lambda calculus, value-based computation, and the mathematical structures of functional programming are granite: older, denser, formed under conditions of higher pressure, and fundamentally more resistant to the acids of time and use.

The Admissibility Program's claim is that admissibility structure is the deepest geological layer of all. Not lambda calculus specifically, not functional programming as a programming paradigm, but the geometry of what can be reached from

where — the structure of admissible transformations across physics, cognition, memory, computation, and civilisation. Lambda calculus is one particularly clean way of making that structure visible in the domain of computation. Payton Jones has spent his career ensuring that the formal admissibility structure of programs is not hidden in a mutable global state somewhere but made visible, legible, and repairable through the type system.

That is not a narrow contribution to programming language design. It is a contribution to the broader project of making admissibility structure legible wherever it lives.

This essay interprets Simon Payton Jones, "Co-Creator of Haskell: Functional Programming, Thinking in Types, Useless Languages," 2025 interview (transcript). All quotations are from that transcript. The Admissibility Program frameworks referenced are documented in Project Map and Intellectual Architecture of the Flyxion Research Program (*Flyxion, 2026*).