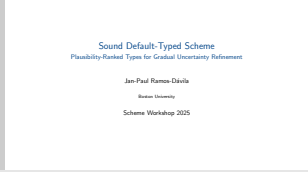
Sound Default-Typed Scheme

Plausibility-Ranked Types for Gradual Uncertainty Refinement

Jan-Paul Ramos-Dávila

Boston University

Scheme Workshop 2025



1. This talk presents a new approach to typing that captures how programmers actually think about their code - distinguishing between what COULD happen and what USUALLY happens. We introduce plausibility-ranked types that enable relaxed but sound guarantees, with two key innovations.
2. First, we provide better error propagation for belief-based assumptions, which is less costly than contracts or refinement types. Second, we enable gradual refinement from qualitative beliefs to quantitative probabilities, maintaining soundness throughout. This is particularly valuable in Scheme where we want to express programmer beliefs without proving them statically or checking them dynamically.

The Gap Between What Could and What Usually Happens

Example:

```
(define (score-latency latencies successes alpha)
  (define n (length latencies))
  (define k (exact-round (* 0.05 n)))  ; trim 5%
  (define trimmed
    (drop (take (sort latencies <) (- n k)) k))
  (define mean (/ (apply + trimmed) (length trimmed)))
  (define fail-ratio (- 1 (/ successes n)))
  (+ (* alpha mean) (* (- 1 alpha) fail-ratio)))
```

Programmer's mental model:

- **Usually**: buckets have 100+ samples, alpha is valid [0,1]
- **Rarely**: edge cases with small n, misconfigured alpha
- **Never**: empty buckets (monitoring ensures this)

But our type systems only capture **always** vs **never**!

The Gap Between What Could and What Usually Happens

Example:

(define (score-latency latencies successes alpha)
(define n (length latencies))
(define k (exact-round (* 0.05 n))) ; trim 5%
(define trimmed
(drop (take (sort latencies <) (- n k)) k))
(define mean (/ (apply + trimmed) (length trimmed)))
(define fail-ratio (- 1 (/ successes n)))
(+ (* alpha mean) (* (- 1 alpha) fail-ratio)))

Programmer's mental model:

- Usually: buckets have 100+ samples, alpha is valid [0,1]
- Rarely: edge cases with small n, misconfigured alpha
- Never: empty buckets (monitoring ensures this)

But our type systems only capture always vs never!

1. This is a load balancer health scoring function - real code from distributed systems. Here's what it does: Given latency measurements from a service instance, it computes a composite health score by combining average latency with failure rate. The key insight: it uses TRIMMED mean to ignore outliers - drops the top and bottom 5% of measurements to avoid skew from occasional spikes or timeouts.
2. The intuition: Service instances collect buckets of timing data. Under normal operation, we have hundreds of samples per bucket, alpha is a tuning parameter (0.0 = only care about latency, 1.0 = only care about failures), and monitoring ensures buckets are never empty. But edge cases exist: small buckets during startup/restarts, misconfigured alpha values. The trimming step is the tricky part - we take the sorted list, drop k elements from each end, then compute the mean. This can fail if the trimmed list becomes empty!
3. Programmers have rich mental models here: "Usually we have 100+ samples so trimming is safe. Rarely we get small buckets and need guards. Never should buckets be empty because monitoring ensures collection." But type systems are binary - they only distinguish ALWAYS (proven safe) vs NEVER (type error). There's no way to express "this usually works" or "this is an edge case I don't want to optimize for." This forces bad tradeoffs: add expensive runtime checks everywhere, prove complex invariants about list lengths, or just hope for the best.

Traditional Approaches Force Bad Tradeoffs

Option 1: Defensive programming (runtime overhead)

```
(define (score-latency-safe latencies successes alpha)
  (cond [(empty? latencies) (error "Empty bucket")]
        [(< (length latencies) 20) (error "Too few samples")]
        [(not (<= 0 alpha 1)) (error "Invalid alpha")]
        [(> successes (length latencies)) (error "Invalid count")]
        [else ... actual algorithm ...]))
```

Option 2: Refinement types (annotation burden)

```
;; Requires: latencies : (Listof-Len>20 Real)
;;           successes : (Integer-Le (length latencies))
;;           alpha : (Real-Between 0 1)
;; Plus proof that trimming preserves non-emptiness...
```

Option 3: Hope and pray (no guarantees)
Our approach: Make assumptions explicit, verify statically, optimize for the common case

Traditional Approaches Force Bad Tradeoffs
Option 1: Defensive programming (runtime overhead)
Option 2: Refinement types (annotation burden)
Option 3: Hope and pray (no guarantees)
Our approach: Make assumptions explicit, verify statically, optimize for the common case

1. Current approaches force us into bad tradeoffs. Defensive programming adds runtime overhead on EVERY call, even though failures are rare. The checks obscure the actual algorithm and hurt performance in the hot path.
2. Refinement types require heavy annotations and proofs. You need to prove that trimming preserves non-emptiness, that your bounds are maintained through operations. This is a huge annotation burden for what should be simple code.
3. Many just skip checks and hope for the best, leading to production failures. Our approach: make assumptions EXPLICIT as ranked types, verify them statically, and generate optimized code for the common case.

Plausibility-Ranked Types

Rank 0: Normal case — $n \geq 20$, alpha in [0,1]

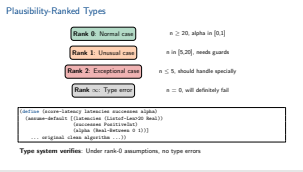
Rank 1: Unusual case — n in [5,20], needs guards

Rank 2: Exceptional case — $n \leq 5$, should handle specially

Rank ∞: Type error — $n = 0$, will definitely fail

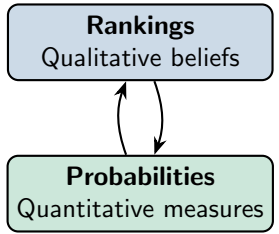
```
(define (score-latency latencies successes alpha)
  (assume-default [(latencies (Listof-Len>20 Real))
                   (successes PositiveInt)
                   (alpha (Real-Between 0 1))]
    ... original clean algorithm ...))
```

Type system verifies: Under rank-0 assumptions, no type errors



1. Our first innovation: plausibility-ranked types. We assign ranks to capture how plausible different scenarios are. Rank 0 is the normal case - what we designed for. Rank 1 is unusual but possible. Rank 2 is exceptional - we don't want to optimize for it. Rank infinity is a type error - definitely wrong.
2. The assume-default form makes assumptions explicit. We're saying: "I'm assuming these properties hold in the normal case." The type system then verifies that UNDER these rank-0 assumptions, there are no type errors. If the assumptions might not hold, we get higher ranks, forcing us to handle edge cases explicitly.
3. This is different from contracts - we're not checking at runtime. Different from refinement types - we're not proving these always hold. We're saying "I believe this is normally true" and the system ensures consistency.

Gradual Refinement via Abstract Interpretation



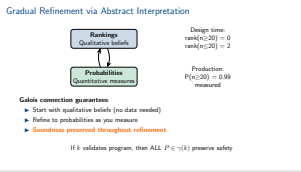
Design time:
rank(n≥20) = 0
rank(n≤20) = 2

Production:
P(n≥20) = 0.99
measured

Galois connection guarantees:

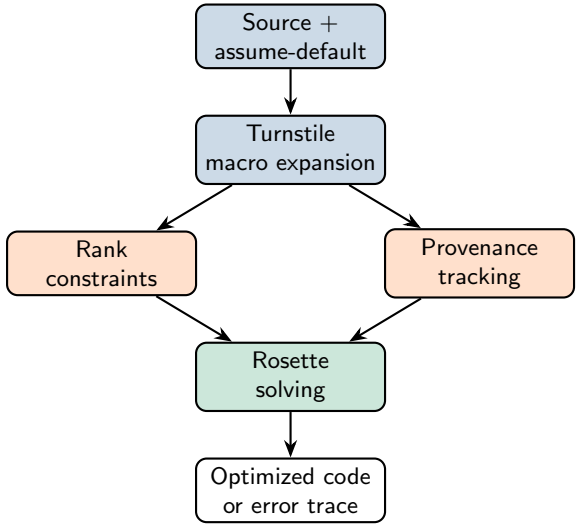
- Start with qualitative beliefs (no data needed)
- Refine to probabilities as you measure
- **Soundness preserved throughout refinement**

If k validates program, then ALL $P \in \gamma(k)$ preserve safety



1. Our second innovation: gradual refinement from qualitative to quantitative via abstract interpretation. We establish a Galois connection between rankings and probabilities. This is the theoretical foundation that makes everything work.
2. At design time, you have qualitative beliefs - "n is usually greater than 20". No numbers needed, just rankings. The abstraction function alpha extracts orderings from probabilities. The concretization gamma returns ALL probability distributions consistent with a ranking.
3. In production, you measure actual distributions. The key theorem: if the ranking validates your program, then ALL probability distributions in its concretization preserve safety. As you refine from qualitative to quantitative, soundness is preserved. This is different from gradual typing - we're gradually refining uncertainty representations, not types.

The Complete Architecture

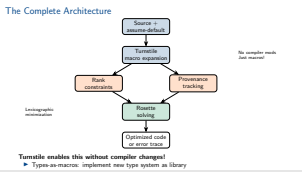


No compiler mods
Just macros!

Lexicographic minimization

Turnstile enables this without compiler changes!

- Types-as-macros: implement new type system as library



1. Here's how we actually build this system using Racket's language-oriented programming. The key insight: Turnstile lets us implement this entire type system WITHOUT modifying the compiler. It's just a library!
2. Source code with assume-default flows into Turnstile, which implements our ranked type system as macros. During macro expansion, we generate two things: rank constraints encoding the type rules, and provenance information tracking where assumptions come from.
3. Both flow into Rosette, which uses Z3 to solve the constraint system with lexicographic minimization - first minimize type errors, then exceptional cases, then unusual cases. If solving succeeds at rank 0, we generate optimized code with no runtime checks. If it fails, we produce an error trace showing WHICH assumption failed and WHERE it came from. This is the power of preserving provenance through macro expansion.

Type Rules with Plausibility Ranks

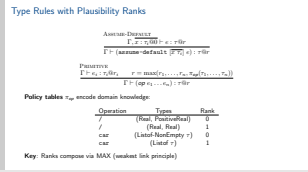
$$\textsc{Assume-Default}\quad \frac{\Gamma, x : \tau_i@0 \vdash e : \tau@r}{\Gamma \vdash (\texttt{assume-default}\ [\overline{x\ \tau_i}]\ e) : \tau@r}$$

$$\textsc{Primitive}\quad \frac{\Gamma \vdash e_i : \tau_i@r_i \qquad r = \max(r_1, \ldots, r_n, \pi_{op}(\tau_1, \ldots, \tau_n))}{\Gamma \vdash (op\ e_1 \ldots e_n) : \tau@r}$$

Policy tables π_{op} encode domain knowledge:

Operation	Types	Rank
/	(Real, PositiveReal)	0
/	(Real, Real)	1
car	(Listof-NonEmpty τ)	0
car	(Listof τ)	1

Key: Ranks compose via MAX (weakest link principle)



1. Here are the core typing rules. The Assume-Default rule is how we introduce plausibility assumptions. We extend the environment with variables at rank ZERO - the normal case. We're formally declaring our beliefs about normal operation. This is the entry point where programmers express "I believe these properties hold normally."
2. The Primitive rule is where composition happens. For any operation, we check all arguments and get their ranks. We also lookup the operation's base rank from the policy table pi. The result rank is the MAXIMUM - the weakest link principle. If ANY subexpression needs exceptional assumptions, the whole expression does. Think of it like a safety chain - you're only as safe as your weakest link.
3. The policy tables are KEY - they encode domain-specific knowledge about operations. Here's how they work: For each primitive operation, we map (operation, input types) to a plausibility rank. Division by PositiveReal is rank 0 - we KNOW it's safe. Division by general Real is rank 1 - might be zero, unusual but needs handling. Car on NonEmpty list is rank 0 - safe. Car on general list is rank 1 - might fail.
4. Programmers implement these PER PROGRAM based on their domain. For our load balancer: we might say (length (Listof-Len¿20)) is rank 0 because we know buckets are big. In a different domain - say parsing - empty lists might be rank 0 (normal case).

Constraint Solving and Error Reporting

Generated constraints for our example:

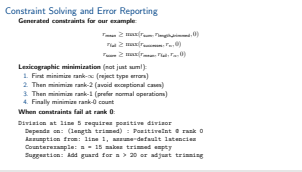
$$r_{\text{mean}} \geq \max(r_{\text{sum}}, r_{\text{length\_trimmed}}, 0)$$
$$r_{\text{fail}} \geq \max(r_{\text{successes}}, r_n, 0)$$
$$r_{\text{score}} \geq \max(r_{\text{mean}}, r_{\text{fail}}, r_\alpha, 0)$$

Lexicographic minimization (not just sum!):

1. First minimize rank-∞ (reject type errors)
2. Then minimize rank-2 (avoid exceptional cases)
3. Then minimize rank-1 (prefer normal operations)
4. Finally minimize rank-0 count

When constraints fail at rank 0:

```
Division at line 5 requires positive divisor
  Depends on: (length trimmed) : PositiveInt @ rank 0
  Assumption from: line 1, assume-default latencies
  Counterexample: n = 15 makes trimmed empty
  Suggestion: Add guard for n > 20 or adjust trimming
```

1. Constraint generation: For each operation, we generate inequalities. When we see division, we look up the policy table: division needs PositiveInt divisor for rank 0. So we generate r-mean ¿= max(r-sum, r-length-trimmed, 0). The result rank is at least as bad as its dependencies.
2. Lexicographic minimization uses priority ordering: First reject any rank-infinity (type errors). Then minimize rank-2 (exceptional). Then rank-1 (unusual). Finally rank-0. This is encoded with exponential weights so one rank-2 outweighs thousands of rank-1s. This ensures we find the most plausible typing, not just any valid typing.
3. Error reporting: When constraints fail at rank 0, we trace back through provenance. Division fails, needs PositiveInt, came from assume-default latencies assumption. Z3 constructs counterexample showing concrete inputs that violate it. Turnstile preserves source locations through macro expansion, letting us show exactly which assumption was too optimistic.

Safe Execution

What you write:

```
(define (score-latency latencies successes alpha)
  (assume-default [(latencies (Listof-Len>20 Real))
                   (successes PositiveInt)
                   (alpha (Real-Between 0 1))]
    ;; Clean algorithm - no defensive checks!
    (define n (length latencies))
    (define k (exact-round (* 0.05 n)))
    (define trimmed (drop (take (sort latencies <) (- n k)) k))
    (define mean (/ (apply + trimmed) (length trimmed)))
    (define fail-ratio (- 1 (/ successes n)))
    (+ (* alpha mean) (* (- 1 alpha) fail-ratio))))
```

What you get:

- Type-checks at rank 0 → Optimized code, no runtime checks
- Fails at rank 0 → Error trace with counterexample
- Compilation time: 100ms for realistic examples

Safe Execution
What you write:
(define (score-latency latencies successes alpha)
(assume-default [(latencies (Listof-Len>20 Real))
(successes PositiveInt)
(alpha (Real-Between 0 1))]
;; Clean algorithm - no defensive checks!
(define n (length latencies))
(define k (exact-round (* 0.05 n)))
(define trimmed (drop (take (sort latencies <) (- n k)) k))
(define mean (/ (apply + trimmed) (length trimmed)))
(define fail-ratio (- 1 (/ successes n)))
(+ (* alpha mean) (* (- 1 alpha) fail-ratio))))
What you get:
Type-checks at rank 0 → Optimized code, no runtime checks
Fails at rank 0 → Error trace with counterexample
Compilation time: 100ms for realistic examples

1. Here's what our system looks like in practice. You write clean code with explicit assumptions via assume-default. The algorithm is uncluttered by defensive checks - it's just the core logic.
2. If type checking succeeds at rank 0, you get optimized code with NO runtime checks in the hot path. Your assumptions are verified to be consistent. If it fails, you get an actionable error showing exactly which assumption is violated and when.
3. The compilation overhead is minimal - around 100ms for realistic examples. This is the time to generate constraints and solve them with Z3. Once compiled, there's zero runtime overhead for the rank-0 path.

Why This Matters

	Contracts	Gradual Types	Our Approach
When checked	Runtime	Boundaries	Compile-time
Performance	Every call	Boundary crossing	Zero overhead
Expressiveness	Any predicate	Type-based	Belief-ranked
Error quality	Runtime failure	Contract violation	Traced assumption
Refactoring	Add checks	Add types	Add assumptions
Philosophy	Defensive	Mixed	Optimistic

Key advantages for Scheme community:

- Preserves an exploratory development style
- No runtime overhead for believed-normal cases
- Gradual path from beliefs to measurements
- Implemented as library via macros (no fork needed!)

Limitations: Module boundaries, higher-order functions need work

Why This Matters

	Contracts	Gradual Types	Our Approach
When checked	Runtime	Boundaries	Compile-time
Performance	Every call	Boundary crossing	Zero overhead
Expressiveness	Any predicate	Type-based	Belief-ranked
Error quality	Runtime failure	Contract violation	Traced assumption
Refactoring	Add checks	Add types	Add assumptions
Philosophy	Defensive	Mixed	Optimistic

Key advantages for Scheme community:
- Preserves an exploratory development style
- No runtime overhead for believed-normal cases
- Gradual path from beliefs to measurements
- Implemented as library via macros (no fork needed!)

Limitations: Module boundaries, higher-order functions need work

1. Let's compare with existing approaches. Contracts check at runtime on every call - overhead even in the hot path. Gradual types check at boundaries - better but still runtime cost. Our approach verifies at compile-time with zero runtime overhead for the normal case.
2. The philosophy differs too. Contracts are defensive - check everything. Gradual types are mixed - some static, some dynamic. We're optimistic - we believe our assumptions and verify consistency, generating fast code for the normal case.
3. This fits Scheme's philosophy perfectly. You can explore and prototype with assumptions, then gradually refine as you learn more. It's implemented as a library using macros - no fork needed! Current limitations: module boundaries and higher-order functions need more work, but the foundation is solid.

Contributions and Next Steps

Next steps:

- Module boundaries with ranked interfaces
- Higher-order functions with latent rankings
- Mining assumptions from existing code
- Integration with Typed Racket's occurrence typing

Vision: A practical theory of "good enough" typing

- Not about proving everything correct
- About making beliefs explicit and checkable
- About optimizing for what usually happens

Code & paper: `github.com/janpaulpl/default-racket`

Contributions and Next Steps

Next steps:
- Module boundaries with ranked interfaces
- Higher-order functions with latent rankings
- Mining assumptions from existing code
- Integration with Typed Racket's occurrence typing

Vision: A practical theory of "good enough" typing
- Not about proving everything correct
- About making beliefs explicit and checkable
- About optimizing for what usually happens

Code & paper: `github.com/janpaulpl/default-racket`

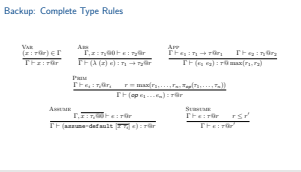
1. To summarize our contributions: We've formalized how to express programmer beliefs about normal versus exceptional cases using plausibility rankings. We've shown how to gradually refine from qualitative beliefs to quantitative probabilities while maintaining soundness via abstract interpretation.
2. We've built a practical implementation using Racket's macro system - no compiler modifications needed. And we provide actionable error messages by tracking provenance through macro expansion.
3. The vision is a practical theory of "good enough" typing. Not about proving everything correct, but about making beliefs explicit and checkable, and optimizing for what usually happens. This is particularly valuable for Scheme's exploratory programming style. The code and paper are available on GitHub.

Backup: Complete Type Rules

$$\textsc{Var}\ \frac{(x : \tau @ r) \in \Gamma}{\Gamma \vdash x : \tau @ r} \qquad \textsc{Abs}\ \frac{\Gamma, x : \tau_1 @ 0 \vdash e : \tau_2 @ r}{\Gamma \vdash (\lambda\ (x)\ e) : \tau_1 \to \tau_2 @ r} \qquad \textsc{App}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau @ r_1 \qquad \Gamma \vdash e_2 : \tau_1 @ r_2}{\Gamma \vdash (e_1\ e_2) : \tau @ \max(r_1, r_2)}$$

$$\textsc{Prim}\ \frac{\Gamma \vdash e_i : \tau_i @ r_i \qquad r = \max(r_1, \ldots, r_n, \pi_{op}(\tau_1, \ldots, \tau_n))}{\Gamma \vdash (op\ e_1 \ldots e_n) : \tau @ r}$$

$$\textsc{Assume}\ \frac{\Gamma, \overline{x : \tau_i @ 0} \vdash e : \tau @ r}{\Gamma \vdash (\texttt{assume-default}\ [\overline{x\ \tau_i}]\ e) : \tau @ r} \qquad \textsc{Subsume}\ \frac{\Gamma \vdash e : \tau @ r \qquad r \le r'}{\Gamma \vdash e : \tau @ r'}$$



1. Complete typing rules for reference. The Var rule does standard lookup. Abs introduces parameters at rank 0 by default. App combines function and argument ranks via max. Subsume allows rank weakening - a rank-0 value can be used where rank-1 is expected.

Backup: Soundness via Galois Connection

Theorem (Conditional Soundness)

If $\vdash e : \tau@0$ *using ranking* k*, then for all* $P \in \gamma(k)$*:*

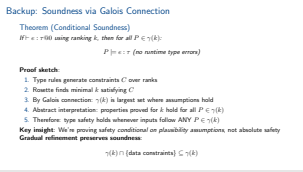
$$P \models e : \tau \text{ (no runtime type errors)}$$

Proof sketch:

1. Type rules generate constraints C over ranks
2. Rosette finds minimal k satisfying C
3. By Galois connection: $\gamma(k)$ is largest set where assumptions hold
4. Abstract interpretation: properties proved for k hold for all $P \in \gamma(k)$
5. Therefore: type safety holds whenever inputs follow ANY $P \in \gamma(k)$

Key insight: We're proving safety *conditional on plausibility assumptions*, not absolute safety
Gradual refinement preserves soundness:

$$\gamma(k) \cap \{\text{data constraints}\} \subseteq \gamma(k)$$



1. Our soundness theorem is conditional, not absolute. If expression e type-checks at rank 0, then it's safe for ALL probability distributions consistent with that ranking. The Galois connection ensures properties proved abstractly hold concretely. As we refine with data, we intersect with constraints, staying within the sound set.

Backup: Lexicographic Optimization Details

Why not just minimize sum of ranks?

```
(define (bad-example x)
  (assume-default [(x Real)]
    (if (zero? x)
        (/ 1 x)     ; Definite error!
        (+ x 1))))  ; Safe operation
```

Sum minimization might accept this with rank 2 + rank 0!
Lexicographic encoding:

$$\text{minimize} \sum_{v \in \text{vars}} 1000^{(3-r_v)}$$

- Rank ∞ gets weight 1000^3
- Rank 2 gets weight 1000^1
- Rank 1 gets weight $1000^0 = 1$
- Rank 0 gets weight 0

One rank-2 outweighs any number of rank-1s!

1. The lexicographic minimization is crucial. Simple sum minimization would accept programs with definite errors by making other parts "more normal". Our encoding uses exponential weights to ensure proper ordering - we reject errors first, then minimize exceptional cases, then unusual cases.