Sound Default-Typed Scheme

Jan-Paul Ramos-Dávila
Boston University
Boston, MA, USA

Abstract

We propose a new approach to typing Scheme programs based on the observation that programmers often have strong beliefs about the "normal" behavior of their code. Rather than forcing a binary choice between static types and runtime checks, we introduce *default typing*, where each program point carries a plausibility-ranked set of types. The highest-ranked type (rank 0) represents what the programmer believes will "almost always" be true, while higher ranks capture increasingly exceptional cases. By leveraging Racket's macro-extensible type system and SMT-based constraint solving, we can verify whether a program type-checks using only the default assumptions. Success yields efficient code with no runtime overhead; failure produces a counterexample showing which assumptions are violated. We provide a precise notion of *conditional soundness*: programs are guaranteed type-safe only when their default assumptions hold at runtime.

CCS Concepts: • **Software and its engineering → Constraint and logic languages**; *Dynamic languages.*

Keywords: Scheme, Racket, type systems, macros, Turnstile, solver-aided languages, plausibility

1 Introduction

The Racket ecosystem has long served as a laboratory for exploring the spectrum between static and dynamic typing. From untyped Racket modules to contracts to Typed Racket, each point in this spectrum offers different tradeoffs. Yet none captures a fundamental aspect of how programmers think: the distinction between what *could* happen and what *usually* happens.

When a Scheme programmer writes `(/ total count)`, they're not thinking "count might be zero." They're thinking "count won't be zero in any reasonable execution." This assumption isn't a proof or even a probabilistic statement: it's a *belief* about normal program behavior. Current type systems force an uncomfortable choice: either prove count is never zero (often impossible), add runtime checks (sacrificing performance), or leave it dynamic (sacrificing safety).

We propose *default typing*, which formalizes these beliefs using plausibility rankings. Each expression gets not one type but a ranked sequence of types. Rank 0 represents the "normal case" the programmer expects. Higher ranks represent increasingly exceptional cases. Our type checker verifies that a program is internally consistent using only rank-0 assumptions. If successful, we generate code with no runtime checks. If not, we show exactly which assumptions conflict.

Using Turnstile [3], we implement our type system as a macro-based DSL that integrates seamlessly with Racket. Using Rosette [4], we leverage SMT solving to find optimal type assignments. The result is a practical system that captures how Scheme programmers actually think about their code.

2 Motivation: A Financial Computing Example

Consider computing moving averages over financial time series:

Listing 1. Moving average computation

```
(define (moving-average prices window)
  (define n (length prices))
  (define (average-window start)
    (define end (+ start window))
    (define subset (take (drop prices start) window))
    (/ (apply + subset) window))
  (build-list (+ (- n window) 1)
              (lambda (i) (average-window i))))
```

This straightforward code embeds several assumptions. The window size must be positive, or division by zero occurs. The price list must have at least `window` elements. The arithmetic must not overflow. In production financial systems, these assumptions are reasonable: market data streams are never empty, window sizes come from validated configuration, and prices stay within reasonable bounds.

Traditional approaches handle these assumptions poorly. Typed Racket would require us to prove statically that window is positive, perhaps changing its type to `Positive-Integer`. But this just pushes the problem to callers, who must now prove they're passing positive values. Contracts check at runtime, adding overhead to every call, which is unacceptable in high-frequency trading. Gradual typing inserts checks at module boundaries, which still incurs runtime cost.

We, instead, choose to allow programmers to state their assumptions explicitly:

Listing 2. Moving average with explicit assumptions

```
(define (moving-average prices window)
  (assume-default
    [(prices (Listof Real))      ; rank 0 assumption
     (window PositiveInteger)]   ; rank 0 assumption
    ...))  ; same body as before
```

The `assume-default` form declares that normally, `prices` is a non-empty list of reals and `window` is positive. Our type checker verifies that under these assumptions, the function body type-checks without any possibility of runtime errors. The generated code runs with no checks.

3 Default Types and Plausibility Rankings

To formalize the notion of "normal" behavior, we adapt plausibility measures from classical AI research on uncertain reasoning [1, 2]. The key insight is that types can be ranked by how plausible they are for a given program point.

Definition 1 (Type Plausibility Measure). *Given a set of types $\mathcal{T}$, a type plausibility measure is a function $\pi : \mathcal{T} \to \mathbb{N} \cup \{\infty\}$ satisfying:*

1. $\pi^{-1}(0) \neq \emptyset$ *(at least one type is maximally plausible)*
2. *If $\tau_1 <: \tau_2$ (subtyping), then $\pi(\tau_1) \geq \pi(\tau_2)$ (supertypes are at least as plausible)*

The second condition captures a crucial design principle. In our framework, lower ranks mean higher plausibility. When `PositiveReal` is a subtype of `Real`, we assign $\pi(\texttt{Real}) = 0$ and $\pi(\texttt{PositiveReal}) = 1$. Why? Because "the value is some real number" is a weaker, more general assumption than "the value is positive." We prefer weaker assumptions as defaults, requiring programmers to explicitly opt into stronger ones.

For our moving average example, consider division. We might assign:

$$\pi_/(\texttt{Real}, \texttt{PositiveReal}) = 0 \quad \text{(normal: positive divisor)}$$
$$\pi_/(\texttt{Real}, \texttt{Real}) = 1 \quad \text{(risky: might be zero)}$$
$$\pi_/(\texttt{Real}, \texttt{Zero}) = 2 \quad \text{(error: definitely zero)}$$

When type-checking `(/ (apply + subset) window)`, the system tries to use the rank-0 rule. This succeeds only if `window` has type `PositiveInteger` at rank 0, which our assumption provides.

3.1 Composing Plausibility Across Expressions

The challenge is determining plausibility for compound expressions. If subexpression e_1 has type τ_1 at rank r_1 and e_2 has type τ_2 at rank r_2, what rank should $(e_1\ e_2)$ have?

We follow the principle of "weakest link": a compound expression is only as plausible as its least plausible part. Formally:

Definition 2 (Plausibility Composition). *For expression e with subexpressions $e_1, \ldots, e_n$, if we can derive $\tau_1, \ldots, \tau_n \vdash e : \tau$ and each e_i has type τ_i at rank r_i, then e has type τ at rank $\max(r_1, \ldots, r_n)$.*

This ensures that if any subexpression requires a non-default assumption (rank > 0), the whole expression inherits that requirement. For our moving average, if we forgot to assume `window` is positive, the division would require rank 1, propagating up through the function.

4 A Scalable Type System

We extend standard typing judgments with plausibility ranks. The judgment $\Gamma \vdash e : \tau@r$ means expression e has type τ at plausibility rank r under context Γ.

Var
$$\frac{(x : \tau@r) \in \Gamma}{\Gamma \vdash x : \tau@r}$$

Abs
$$\frac{\Gamma, x : \tau_1@0 \vdash e : \tau_2@r}{\Gamma \vdash (\texttt{lambda}\ (x)\ e) : \tau_1 \to \tau_2@r}$$

App
$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau@r_1 \qquad \Gamma \vdash e_2 : \tau_1@r_2}{\Gamma \vdash (e_1\ e_2) : \tau@\max(r_1, r_2)}$$

Subsume
$$\frac{\Gamma \vdash e : \tau@r \qquad r \leq r'}{\Gamma \vdash e : \tau@r'}$$

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

Prim
$$\frac{op : \forall\bar{\alpha}.\tau_1 \times \cdots \times \tau_n \to \tau \qquad \Gamma \vdash e_i : \tau_i[\bar{\sigma}/\bar{\alpha}]@r_i \qquad r = \max(r_1, \ldots, r_n, \pi_{op}(\tau_1[\bar{\sigma}/\bar{\alpha}], \ldots))}{\Gamma \vdash (op\ e_1 \ldots e_n) : \tau[\bar{\sigma}/\bar{\alpha}]@r}$$

Figure 1. Typing rules with plausibility ranks (simplified)

The key innovation is the Prim rule, which handles all primitive operations uniformly. Each primitive op has an associated plausibility measure π_{op} that maps input type tuples to minimum ranks. For example:

$\pi_/(\texttt{Real}, \texttt{PositiveReal}) = 0$	Safe division
$\pi_/(\texttt{Real}, \texttt{Real}) = 1$	Might divide by zero
$\pi_{\texttt{car}}((\texttt{Listof}\ \tau)) = 0$	Non-empty list
$\pi_{\texttt{car}}((\texttt{Listof}^?\tau)) = 1$	Might be empty
$\pi_+(\texttt{Integer}, \texttt{Integer}) = 0$	Usually no overflow
$\pi_+(\texttt{BigInt}, \texttt{BigInt}) = 0$	Never overflows

This approach scales to Racket's full primitive set by defining appropriate plausibility measures. The Subsume rule enables rank flexibility: a rank-0 value can be used where rank-1 is expected (normal cases are special cases of exceptional ones).

5 From Types to Constraints

Type checking generates constraints over rank variables rather than making binary decisions.

5.1 Constraint Generation

During macro expansion, Turnstile traverses the program generating constraints. For our moving average example, the expression `(/ (apply + subset) window)` generates:

$$r_{sum} \geq \max(r_{apply}, r_{+}, r_{subset}) \quad (1)$$

$$r_{result} \geq \max(r_{sum}, r_{window}, \pi_{/}(\tau_{sum}, \tau_{window})) \quad (2)$$

The first constraint says the sum's rank is at least the maximum of its components' ranks. The second says the division's rank depends on both operands' ranks and their types. If $\tau_{window} = \texttt{PositiveInteger}$, then $\pi_{/}(\texttt{Real}, \texttt{PositiveInteger}) = 0$. But if $\tau_{window} = \texttt{Integer}$, then $\pi_{/}(\texttt{Real}, \texttt{Integer}) = 1$.

5.2 Why Lexicographic Minimization?

Simply minimizing the sum of all ranks fails catastrophically. Consider:

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

A naive solver might assign rank 0 to the else branch and rank 2 to the error, achieving sum 2. But this accepts a program with a definite error!

Instead, we use lexicographic minimization: first minimize the count of rank-2 (error) assignments, then rank-1 (exceptional), then rank-0. This is encoded as minimizing:

$$\sum_{v \in \text{vars}} 2^{(n-r_v)} \cdot [r_v = k]$$

where $[r_v = k]$ is 1 if variable v has rank k, and 0 otherwise. The exponential weighting ensures that one rank-2 assignment outweighs any number of rank-1 assignments. For our bad example, this forces the type checker to report the definite division by zero.

5.3 Rosette Integration

We implement constraint solving using Rosette's symbolic execution:

Listing 3. Simplified Rosette integration

```
(define-symbolic-struct ranking
  ([vars (hash/c symbol? integer?)]))

(define (solve-constraints assumptions body)
  (define r (ranking (make-hash)))
  ; Add assumption constraints
  (for ([a assumptions])
    (assert (= (hash-ref r.vars (car a)) 0)))
  ; Add body constraints
  (define body-constraints (generate-constraints body r))
  (for ([c body-constraints]) (assert c))
  ; Solve with lexicographic objective
  (optimize #:minimize
    (sum-weighted-ranks r.vars)
    #:guarantee (verify r)))
```

For the 200-line moving average example, this typically solves in under 100ms on modern hardware, making it practical for interactive development.

6 Design Decisions and Future Directions

6.1 Trade-offs

Explicit over inferred assumptions. We require programmers to state assumptions explicitly rather than inferring them. While inference might seem convenient, it obscures programmer intent and can hide bugs. If the system inferred that division operands might be zero, it would silently accept potentially buggy code. Explicit assumptions serve as checked documentation.

Module boundaries. Our current prototype checks each module independently, which can miss cross-module assumption violations. We envision module contracts that include plausibility ranks:

```
(provide/contract/default
  [process-data
   (->i ([data (listof real?)]
         [window positive-integer?])
        #:pre/default (data window)
        (> (length data) window)
        [result (listof real?)])])
```

When whole-program analysis is impractical we fall back on conservative ranks: imported values default to rank 1, local code may still enjoy rank 0. Empirically, most performance hotspots live in single modules anyway. Future work explores "ranked" interface files akin to Typed Racket's .Rktd sigs.

Higher-order functions. These remain challenging. When mapping a function over a list of positive numbers, how do we ensure the function preserves positivity? Effect systems like those in Koka [10] suggest a path forward, tracking not just types but type transformations.

Conditional soundness. We provide a weaker guarantee than traditional type systems: if the program type-checks at rank 0 and runtime inputs satisfy rank-0 assumptions, execution is type-safe. This is formalized as:

$$\Gamma \vdash_0 e : \tau \wedge \sigma \models \Gamma \Rightarrow \langle e, \sigma \rangle \nrightarrow \mathit{error}$$

This conditional guarantee fits many practical scenarios where full verification is impossible but common-case correctness is crucial.

Why not just use contracts? Contracts remain the right tool for rich cross-module invariants. But contracts charge an overhead at every call site, which is untenable in hot paths (finance, graphics pipelines, numerics). Default typing shifts that cost into compile time and only for the normal case; exceptional paths may still fall back to contracts if desired.

6.2 Going forward

There are a handful of immediate areas for future work. We can start with the question: what if the programmer is not sure when to explicitly call `assume-default` blocks? To reduce friction, we propose *assumption mining*: a Rosette-backed profiler executes the program on representative inputs, clusters observed types, and suggests plausible rank-0 candidates.

In addition, we are intent on mechanizing our soundness proof, integrating with Typed Racket's occurrence typing, and conducting empirical studies on real Racket codebases. We're particularly interested in whether programmers' actual assumptions align with what our system can verify.

7 Related Work

Default typing occupies a unique position in the landscape of type systems for dynamic languages. Soft typing [5, 6] pioneered static analysis for Scheme but focused on finding definite errors. We focus on verifying programmer assumptions about normal behavior. Where soft typing is conservative (rejecting only definite errors), we're optimistic (accepting only programs correct under stated assumptions).

Gradual typing [7, 8] allows mixing typed and untyped code but requires runtime checks at boundaries. We reject any program needing runtime checks, forcing all assumptions to be explicit and statically verified.

Occurrence typing in Typed Racket [8] refines types based on runtime tests. We could extend our system to refine plausibility ranks after type tests, potentially enabling more rank-0 assumptions in branches.

Refinement types [9] prove precise properties but require significant annotation effort. Default typing provides a lighter-weight alternative for cases where full verification is unnecessary. A hybrid system could use refinement types for critical properties and default types for performance optimizations.

8 Conclusion

Default typing attempts to formalize a fundamental aspect of how programmers think: the distinction between what could happen and what usually happens. By integrating plausibility rankings into Racket's extensible type system, we enable efficient code generation for common cases while maintaining safety guarantees.

Our approach is particularly suited to domains where programmers have strong beliefs about normal behavior: financial computing (non-zero values), scientific computing (non-singular matrices), and systems programming (successful allocations). It's less suitable for adversarial contexts where assumptions may be violated deliberately.

The integration with Turnstile and Rosette demonstrates the power of Racket's language-oriented programming approach. What would require compiler modifications in other languages becomes a library in Racket. This opens possibilities for domain-specific variations: different plausibility measures for different problem domains.

We see default typing not as a replacement for existing approaches but as a new point in the design space. One that acknowledges the reality of how programmers reason about their code while providing machine-checkable guarantees. In the spirit of the Scheme community's tradition of thoughtful language design, we offer this approach as a tool for capturing programmer intent more faithfully.

References

[1] Halpern, J. Y., & Friedman, N. (1995). Plausibility measures and default reasoning. In *AAAI* (pp. 1297–1304).

[2] Halpern, J. Y. (2003). *Reasoning about uncertainty*. MIT Press.

[3] Chang, S., Knauth, A., & Greenman, B. (2017). Type systems as macros. In *POPL* (pp. 694–705).

[4] Torlak, E., & Bodik, R. (2014). A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI* (pp. 530–541).

[5] Cartwright, R., & Fagan, M. (1991). Soft typing. In *PLDI* (pp. 278–292).

[6] Wright, A. K., & Cartwright, R. (1994). A practical soft type system for Scheme. In *LFP* (pp. 250–262).

[7] Siek, J. G., & Taha, W. (2006). Gradual typing for functional languages. In *Scheme Workshop*.

[8] Tobin-Hochstadt, S., & Felleisen, M. (2008). The design and implementation of Typed Scheme. In *POPL* (pp. 395–406).

[9] Vazou, N., et al. (2014). Refinement types for Haskell. In *ICFP* (pp. 269–282).

[10] Leijen, D. (2014). Koka: Programming with row polymorphic effect types. In *MSFP*.