The Spectrum of Soundness and Performance

Anonymous Author(s)

Abstract

Gradual typing provides software developers with the means to re-factor an untyped code base into a typed one. As this refactoring proceeds, the code base becomes a mixture of typed and untyped code. In this mixed-typed world, developers may wish to rely on the soundness of the type system. Industrial variants of gradual typing, however, guarantee only an extremely limited form of soundness, and as recently shown, the soundness of academic implementations poses a serious performance bottleneck, which limits the deployability of mixed-typed software.

In this paper, we develop the novel idea of a spectrum of type soundness and demonstrate that a form of whole-program, non-compositional form of soundness vastly improves the performance of Typed Racket, the most mature implementation of sound gradual typing. Specifically, we develop a series of theoretical models of gradual typing, starting with a classically sound gradual core language. Furthermore, we explain how to modify Typed Racket's implementation and present the results of measuring this revised implementation on the Takikawa-Greenman benchmarks.

1 Three Flavors of Migratory Typing

Over the past two decades, software developers have migrated from the world of C++ and Java to the broad world of untyped languages: JavaScript, Perl, Python, Ruby (on Rails), and many others. About one decade ago, they discovered the cost of using languages without type systems. In response, academic researchers simultaneously proposed gradual typing [23, 24] and migratory typing [28, 30], both of which allow developers to mix typed into untyped code. Using such systems, developers can incrementally equip a code base with types and migrate it to the typed world.

In theory, these type systems provide developers with an ideal pathway to equip a code base with types. To begin with, developers can add types wherever needed, explicitly stating (and checking) invariants that will help future readers to understand the code. Better still, the software system keeps functioning, regression test suites remain applicable, and the added types may just reveal some long-hidden obscure errors; in contrast, a whole-sale port to a (completely) statically typed language merely forces developers to go through the whole development process again—without improving the well-tested system. All this theory can work only if the performance of the mixed-type code base remains acceptable. As Takikawa et al. have recently shown [27], however, the performance is definitely *not* acceptable when the underlying type system is *sound*. By comparison, industrial implementations of gradual typing come without performance problems because they do not insist on type soundness.

From a type-theoretic perspective, academic migratory type systems use a type-directed "natural" embedding of typed code into an untyped context [18]. Roughly speaking, when an untyped value mixes with typed code, it gets "type checked" at run time; viewed from the typed perspective, a typed value gets protected with a checking wrapper when it flows into an untyped context. All of these checks add a significant overhead when mixed-typed code is run; the performance improves only when (almost) the entire code base is equipped with types. Industrial variants of these type systems use an "erasure embedding." The result is code that does not detect when a run-time state violates the type invariant of the source code; indeed, it may never signal such a violation; but, it relies on the existing run-time checks of the underlying untyped language to prevent type violations from turning into segmentation faults (of unsafe languages such as C++).

This paper contributes (1) several models of migratory type systems that sit between these two extremes (see sec. 3). The development is inspired by Reticulated Python [31], a pre-processor implementation of a gradual type system for Python. We state theorems that demonstrate to what extent each model is type-sound. (2) We explain how to implement the most promising variant in Typed Racket so that we can compare "apples with apples" (see sec. 4). And finally, we present the results of applying Takikawa et al.'s method to this implementation (see sec. 5). The speed-up improvements are dramatic; typically an order-of-magnitude improvement. We thus consider these results a first step toward the creation of a feasible, sound migratory type system. Section 2 starts the paper with a presentation of the background, and section 6 explains the context in some detail.

2 The Multi-Language Approach

Given a dynamically typed language, the design of a migratory type system poses two problems. The first one concerns the existing programming idioms. To avoid the need for

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

large-scale rewrites, the new type system ought to support existing idioms, which may necessitate the invention of new type system elements. The second one concerns the boundary between the typed and the untyped portions of code repositories. As mentioned, we cannot assume that developers equip all their "untyped code" with the necessary type annotations so mixed-typed programs will exist. This paper focuses on the second problem, because it is the dominant source of performance problems and deserves more attention than it has received initially.

Our novel take on this "boundary problem" is to understand it as a multi-language problem in the spirit of Matthews and Findler's idea [18]. From this perspective, a migratory typing system for a dynamically-typed host language $\mathbf{L_D}$ adds two key pieces: (1) a statically-typed language $\mathbf{L_S}$, and (2) a typed foreign-function interface (FFI) between the languages. The language $\mathbf{L_S}$ is basically like the host language with syntax for explicit type annotations. The foreign-function interface (FFI) is typically part of a runtime system that monitors interactions between statically-typed and dynamically-typed values. The FFI *introduces* the boundary (and therefore run-time) checks that ensure type soundness.

From the literature on multi-language semantics we know that an FFI requires a well-specified embedding of values from one language in the other and that this embedding aims to provide a soundness guarantee. Since $\mathbf{L_D}$ and $\mathbf{L_S}$ share values, value conversion is not a problem for us [3, 13].

Note We must make a choice concerning which values may cross these special boundaries. To keep the boundaries as inexpensive as possible, we might wish to restrict the set of FFI values to just numbers and characters, i.e., small, "flat" values. To facilitate the migration from the untyped world to the typed one, we ought to allow such boundaries to show up wherever developers need them, which implies that all kinds of values may cross the boundaries. All practical forms of migratory and gradual type system prefer the second choice, and so do we in this paper. **End**

An *embedding* in this sense may consist of static and dynamic components. On the static end, the multi-language may add expression and value forms, as well as typing rules for the new additions. At a minimum, the extension must include so-called *boundary terms* to draw a line between code from either source language; we use $(\mathsf{dyn}\ \tau\ e)$ and $\mathsf{stat}\ \tau\ e$. A dyn expression embeds a dynamically-typed expression e into a statically-typed context that expects a value of type τ. A stat expression embeds a statically-typed expression e of type τ into a dynamically-typed context.

On the dynamic end, the multi-language needs runtime support for any new values and for boundary terms. For boundary terms, we require type-directed reduction strategies for moving value forms across boundary terms.

The following section develops several different strategies in the context of several lambda-calculus based language models. Each embedding strategy comes with soundness

Common Syntax

$$
\begin{array}{lll}
e & = x \mid v \mid e\ e \mid \langle e, e\rangle \mid op^1\ e \mid op^2\ e\ e \\
v & = i \mid \langle v, v\rangle \\
i & = 0 \mid 1 \mid -1 \mid \ldots \\
op^1 & = \mathsf{fst} \mid \mathsf{snd} \\
op^2 & = \mathsf{sum} \mid \mathsf{quotient} \\
A & = e \mid \mathsf{BoundaryError} \mid \mathsf{TagError} \\
E & = [\,] \mid E\,e \mid v\,E \mid \langle E, e\rangle \mid \langle v, E\rangle \mid \\
 & \quad\ op^1\ E \mid op^2\ E\ e \mid op^2\ v\ E
\end{array}
$$

$\boxed{\delta(op^1, v) = A}$

$$
\begin{array}{ll}
\delta(\mathsf{fst}, \langle v_0, v_1\rangle) & = v_0 \\
\delta(\mathsf{snd}, \langle v_0, v_1\rangle) & = v_1
\end{array}
$$

$\boxed{\delta(op^2, v, v) = A}$

$$
\begin{array}{ll}
\delta(\mathsf{sum}, i_0, i_1) & = i_0 + i_1 \\
\delta(\mathsf{quotient}, i_0, 0) & = \mathsf{BoundaryError} \\
\delta(\mathsf{quotient}, i_0, i_1) & = \lfloor i_0/i_1 \rfloor \\
\quad \text{if } i_1 \neq 0 &
\end{array}
$$

$\boxed{e \mapsto^{E}_{\mathrm{R}} A}$

$$
\begin{array}{ll}
E[e] \mapsto^{E}_{\mathrm{R}} E[e'] & \text{if } e\ \mathrm{R}\ e' \\
E[e] \mapsto^{E}_{\mathrm{R}} A & \text{if } e\ \mathrm{R}\ A \text{ and } A \notin e
\end{array}
$$

Figure 1: Common syntax and semantics

benefits and performance costs, which just these choices explain. Some choices explain existing industrial choices, while others explain academic choices.

3 Five Embeddings

The goal of a type-directed embedding is to describe how three classes of values may cross language boundaries: (1) values of a base type, (2) finite values of a parameterized type, and (3) infinite values of a parameterized type.[1] As representative examples, we use integers, pairs, and (anonymous, single-argument) functions. Scaling an embedding to accomodate other types and values is usually straightforward, as we discuss in section 4.

Figure 1 introduces the common syntactic and semantic notions. Expressions e include variables, value forms, and the application of a function or primitive operation to arguments. The unary primitive operations fst and snd are projection functions for pair values; the binary primitives sum and quotient are integer arithmetic operators.

The semantics of the primitive operations is given by the partial δ function. In a real language, these primitives would be implemented by a runtime system that manipulates the machine representation of values. As such, we treat calls to δ as cross-language function calls. The result of such a function call is either a value, a token indicating a cross-language boundary error, or undefined behavior.

Undefined behavior due to δ is a categorical evil. The baseline soundness requirement for our models is that they rule out programs that can lead to undefined behavior.

[1] By finite and infinite, we refer to these values' observable behaviors.

Language $\mathbf{L_D}$

$v = \ldots \mid \lambda x.\, e$

$\Gamma = \cdot \mid x, \Gamma$

$\boxed{\Gamma \vdash_D e}$

$$\frac{x \in \Gamma}{\Gamma \vdash_D x} \qquad \frac{x, \Gamma \vdash_D e}{\Gamma \vdash_D \lambda x.\, e} \qquad \frac{}{\Gamma \vdash_D i} \qquad \frac{\Gamma \vdash_D e_0 \quad \Gamma \vdash_D e_1}{\Gamma \vdash_D \langle e_0, e_1 \rangle}$$

$$\frac{\Gamma \vdash_D e_0 \quad \Gamma \vdash_D e_1}{\Gamma \vdash_D e_0\ e_1} \qquad \frac{\Gamma \vdash_D e}{\Gamma \vdash_D op^1\ e} \qquad \frac{\Gamma \vdash_D e_0 \quad \Gamma \vdash_D e_1}{\Gamma \vdash_D op^2\ e_0\ e_1}$$

$\boxed{e \ \mathrm{D}\ A}$

$(\lambda x.\, e)\ v$ D $e[x \leftarrow v]$

$v_0\ v_1$ D TagError if $v_0 \sim i$ or $v_0 \sim \langle v, v' \rangle$

$op^1\ v$ D A if $A = \delta(op^1, v)$

$op^1\ v$ D TagError otherwise

$op^2\ v_0\ v_1$ D A if $A = \delta(op^2, v_0, v_1)$

$op^2\ v_0\ v_1$ D TagError otherwise

$\boxed{e \to_{\mathrm{D}}^{*} A}$ reflexive, transitive closure of $\mapsto_{\mathrm{D}}^{E}$

Figure 2: Dynamic Typing

Other components in figure 1 help define semantics. An answer A is either an expression or an error token. Evaluation contexts E impose a left-to-right order of evaluation for the extension of this base with call-by-value functions. Lastly, the meta-function $\mapsto_{\mathrm{R}}^{E}$ lifts a notion of reduction [2] over evaluation contexts in a way that detects and propagates errors [10].

3.1 Source Languages

The language $\mathbf{L_D}$ defined in figure 2 is dynamically-typed. An $\mathbf{L_D}$ expression e is well-formed according to the typing judgment $\Gamma \vdash_D e$ if it contains no free variables. The notion of reduction D defines the semantics of well-formed expressions; in essence, it reduces a valid application of values to a normal answer and maps an invalid application to a token representing a type-tag error [17].

The language $\mathbf{L_S}$ in figure 3 is a statically-typed counterpart to $\mathbf{L_D}$. Types in $\mathbf{L_S}$ describe four interesting classes of $\mathbf{L_D}$ values: integers, natural numbers, pairs, and functions. The type for natural numbers is representative of subset types that do not have a matching low-level type tag [8]. Migratory typing systems must accomodate such types, because they have emerged in dynamically-typed programming languages. An $\mathbf{L_S}$ expression e is well-typed if $\Gamma \vdash_S e : \tau$ can be derived using the rules in figure 3 for some type τ.[2] The purpose of this typing judgment is to guarantee that all application forms apply a function and all primitive operations receive

[2]These typing rules are not syntax directed; see the PLT Redex models in our artifact for a syntax-directed implementation.

Language $\mathbf{L_S}$

$v = \ldots \mid \lambda(x{:}\tau).\, e$

$\tau = \mathsf{Int} \mid \mathsf{Nat} \mid \tau \times \tau \mid \tau \to \tau$

$\Gamma = \cdot \mid (x{:}\tau), \Gamma$

$\boxed{\Gamma \vdash_S e : \tau}$

$$\frac{(x{:}\tau) \in \Gamma}{\Gamma \vdash_S x : \tau} \qquad \frac{(x{:}\tau_d), \Gamma \vdash_S e : \tau_c}{\Gamma \vdash_S \lambda(x{:}\tau_d).\, e : \tau_d \to \tau_c} \qquad \frac{i \in \mathbb{N}}{\Gamma \vdash_S i : \mathsf{Nat}}$$

$$\frac{}{\Gamma \vdash_S i : \mathsf{Int}} \qquad \frac{\Gamma \vdash_S e_0 : \tau_0 \quad \Gamma \vdash_S e_1 : \tau_1}{\Gamma \vdash_S \langle e_0, e_1 \rangle : \tau_0 \times \tau_1} \qquad \frac{\Gamma \vdash_S e_0 : \tau_d \to \tau_c \quad \Gamma \vdash_S e_1 : \tau_d}{\Gamma \vdash_S e_0\ e_1 : \tau_c}$$

$$\frac{\Gamma \vdash_S e_0 : \tau_0 \quad \Delta(op^1, \tau_0) = \tau}{\Gamma \vdash_S op^1\ e_0 : \tau} \qquad \frac{\Gamma \vdash_S e_0 : \tau_0 \quad \Gamma \vdash_S e_1 : \tau_1 \quad \Delta(op^2, \tau_0, \tau_1) = \tau}{\Gamma \vdash_S op^2\ e_0\ e_1 : \tau} \qquad \frac{\Gamma \vdash_S e : \tau' \quad \tau' <: \tau}{\Gamma \vdash_S e : \tau}$$

$\boxed{\Delta(op^1, \tau) = \tau}$

$\Delta(\mathsf{fst}, \tau_0 \times \tau_1) = \tau_0$

$\Delta(\mathsf{snd}, \tau_0 \times \tau_1) = \tau_1$

$\boxed{\Delta(op^2, \tau, \tau) = \tau}$

$\Delta(op^2, \mathsf{Nat}, \mathsf{Nat}) = \mathsf{Nat}$

$\Delta(op^2, \mathsf{Int}, \mathsf{Int}) = \mathsf{Int}$

$\boxed{\tau <: \tau}$

$$\frac{}{\mathsf{Nat} <: \mathsf{Int}} \qquad \frac{\tau_d' <: \tau_d \quad \tau_c <: \tau_c'}{\tau_d \to \tau_c <: \tau_d' \to \tau_c'} \qquad \frac{\tau_0 <: \tau_0' \quad \tau_1 <: \tau_1'}{\tau_0 \times \tau_1 <: \tau_0' \times \tau_1'}$$

$\boxed{e \ \mathrm{s}\ A}$

$(\lambda(x{:}\tau_d).\, e)\ v$ s $e[x \leftarrow v]$

$op^1\ v$ s $\delta(op^1, v)$

$op^2\ v_0\ v_1$ s $\delta(op^2, v_0, v_1)$

$\boxed{e \to_{\mathrm{s}}^{*} A}$ reflexive, transitive closure of $\mapsto_{\mathrm{s}}^{E}$

Figure 3: Static Typing

arguments for which δ is defined. If this is true, then the notion of reduction s is defined for all well-typed expressions.

Both languages are sound in a precise sense. For $\mathbf{L_D}$, soundness means that the evaluation of any well-formed expression either produces a valid answer or runs forever. Expressions cannot send the evaluator to an undefined state.

Theorem ($\mathbf{L_D}$ *soundness*) If $\vdash_D e$ then either:

- $e \to_{\mathrm{D}}^{*} v$
- $e \to_{\mathrm{D}}^{*}$ TagError
- $e \to_{\mathrm{D}}^{*}$ BoundaryError
- e diverges

Proof Sketch: By progress and preservation lemmas [34] for the $\vdash_D$ relation. In other words, $e \to_{\mathrm{D}} A$ is defined for all well-formed expressions e and if it maps e to another expression, then the result is well-formed. □

The analogous soundness theorem for $\mathbf{L_S}$ guarantees that evaluation via $\to_{\mathrm{s}}^{*}$ never leads to undefined behavior even

Language $\mathbf{L_M}$ inherits Common Syntax:

$e = \ldots \mid \mathsf{dyn}\ \tau\ e \mid \mathsf{stat}\ \tau\ e$
$v = \ldots \mid \lambda x.\, e \mid \lambda(x{:}\tau).\, e$
$\tau = \mathsf{Nat} \mid \mathsf{Int} \mid \tau \times \tau \mid \tau \to \tau$
$\Gamma = \cdot \mid x, \Gamma \mid (x{:}\tau), \Gamma$

$\Gamma \vdash e$ inherits $\Gamma \vdash_D e$: $\quad$ $\Gamma \vdash e : \tau$ inherits $\Gamma \vdash_S e : \tau$:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{stat}\ \tau\ e} \qquad \frac{\Gamma \vdash e}{\Gamma \vdash \mathsf{dyn}\ \tau\ e : \tau}$$

$\mathrm{R} \bowtie_E \mathrm{R}'$ $= (\to_{\mathrm{R}}, \to_{\mathrm{R}'})$ where:

$E[\mathsf{dyn}\ \tau\ e] \to_{\mathrm{R}} E[\mathsf{dyn}\ \tau\ e']$ if $e \to_{\mathrm{R}'} e'$
$E[\mathsf{dyn}\ \tau\ e] \to_{\mathrm{R}} A$ if $e \to_{\mathrm{R}'} A$ and $A \notin e'$
$e \to_{\mathrm{R}} A$ if $e \mapsto^{E}_{\mathrm{R}} A$

$E[\mathsf{stat}\ \tau\ e] \to_{\mathrm{R}'} E[\mathsf{stat}\ \tau\ e']$ if $e \to_{\mathrm{R}} e'$
$E[\mathsf{stat}\ \tau\ e] \to_{\mathrm{R}'} A$ if $e \to_{\mathrm{R}} A$ and $A \notin e'$
$e \to_{\mathrm{R}'} A$ if $e \mapsto^{E}_{\mathrm{R}'} A$

Figure 4: Multi-Language $\mathbf{L_M}$

though the reduction relation calls the partial δ function and never raises tag errors. Additionally, we can prove that evaluation preserves types; if if an expression has type τ and evaluates to a value v, then v also has type τ. This enhancement allows programmers to use static type information to reason about the run-time behavior of programs.

Theorem ($\mathbf{L_S}$ *type soundness*) If $\vdash_S e : \tau$ then either:

- $e \to^*_{\mathrm{S}} v$ and $\vdash_S v$
- $e \to^*_{\mathrm{S}}$ BoundaryError
- e diverges

Proof Sketch: By progress and preservation of the $\vdash_S \cdot : \tau$ relation. (The only boundary error is division by zero.) □

3.2 Multi-Language Syntax

Figure 4 defines the syntax and typing rules of a multi-language based on $\mathbf{L_D}$ and $\mathbf{L_S}$. The multi-language $\mathbf{L_M}$ recursively extends the common syntax in figure 1 with boundary expressions, combined value forms, and combined type contexts. This language comes with two mutually-recursive typing judgments: a well-formedness judgment $\Gamma \vdash e$ for dynamically-typed expressions and a type-checking judgment $\Gamma \vdash e : \tau$ for statically-typed expressions. Note that the typing rules prevent a dynamically-typed expression from directly referencing a statically-typed variable, and vice-versa. Cross-language references must go through a dyn or stat boundary expression, depending on the context.

By intention, the definition of $\mathbf{L_M}$ does not include a semantics. The rest of this section introduces five alternative semantics, each with unique tradeoffs. Figure 4 does include a meta-function that defines two mutually-recursive reduction relations from two notions of reduction. We use $\mathrm{R} \bowtie_E \mathrm{R}'$, pronounced "R-static bowtie-E R′-dynamic", to build: (1) a reduction relation $\to_{\mathrm{R}}$ for statically-typed expressions, and (2) a reduction relation $\to_{\mathrm{R}'}$ for dynamically-typed expressions. Informally, $\to_{\mathrm{R}}$ applied to a statically-typed expression e applies R provided e is not currently evaluating a boundary term; otherwise $\to_{\mathrm{R}}$ dispatches to the analogous $\to_{\mathrm{R}'}$ and the two flip-flop for nested boundaries. The payoff of this technical machinery is that a statically-typed term e cannot step via $\to_{\mathrm{R}}$ to a type-tag error if e does not embed dynamically-typed code, which facilitates the proof of the soundness theorems.

Language $\mathbf{L_E}$ inherits $\mathbf{L_M}$:

$E = \ldots \mid \mathsf{stat}\ \tau\ E \mid \mathsf{dyn}\ \tau\ E$

$\Gamma \vdash_E e$ inherits $\Gamma \vdash_D e$:

$$\frac{x, \Gamma \vdash_E e}{\Gamma \vdash_E \lambda(x{:}\tau).\, e} \qquad \frac{\Gamma \vdash_E e}{\Gamma \vdash_E \mathsf{dyn}\ \tau\ e} \qquad \frac{\Gamma \vdash_E e}{\Gamma \vdash_E \mathsf{stat}\ \tau\ e}$$

$e\ \mathrm{E}\ A$ extends $e\ \mathrm{D}\ A$:

$(\lambda(x{:}\tau_d).\, e)\ v\ \mathrm{E}\ e[x \leftarrow v]$
$\mathsf{dyn}\ \tau\ v\ \mathrm{E}\ v$
$\mathsf{stat}\ \tau\ v\ \mathrm{E}\ v$

$e \to^*_{\mathrm{E}} A$ reflexive, transitive closure of $\mapsto^{E}_{\mathrm{E}}$

Figure 5: Erasure Embedding

3.3 The Erasure Embedding

Intuitively, we can create a multi-language that avoids undefined behavior but does not guarantee any kind of type soundness in two easy steps. First, let statically-typed values and dynamically-typed values freely cross boundary terms. Second, base the evaluator on the dynamically-typed notion of reduction.

Figure 5 specifies this *erasure semantics* for the $\mathbf{L_M}$ language. The notion of reduction E extends the dynamically-typed reduction to handle type-annotated functions and boundary expressions; it ignores the types. Its definition relies on an extension of evaluation contexts to allow reduction under boundaries and take the appropriate closure of E. The typing judgment $\Gamma \vdash_E e$ recursively extends the notion of a well-formed program to ignore any type annotations. Using $\Gamma \vdash_E \cdot$ as a run-time invariant, we can state a soundness theorem for $\mathbf{L_E}$:

Theorem ($\mathbf{L_E}$ *term safety*) If $\vdash e : \tau$ then $\vdash_E e$ and either:

- $e \to^*_{\mathrm{E}} v$ and $\vdash_E v$
- $e \to^*_{\mathrm{E}}$ TagError
- $e \to^*_{\mathrm{E}}$ BoundaryError
- e diverges

Clearly, the erasure embedding is unsound with respect to static types. One can easily build a well-typed expression that reduces to a value with a completely different type. For

$\boxed{\text{Language } \mathbf{L_N}}$ inherits $\mathbf{L_M}$:

$v = \ldots \mid \mathsf{mon}\,(\tau \to \tau)\,v$

$\boxed{\Gamma \vdash_N e}$ inherits $\Gamma \vdash e$:

$$\frac{}{\Gamma \vdash_N \mathsf{mon}\,(\tau_d \to \tau_c)\,v}$$

$\boxed{e\ \mathrm{D}_N\ A}$ extends $e\ \mathrm{D}\ A$:

$(\mathsf{mon}\,\tau_d \to \tau_c\,v_f)\,v\ \mathrm{D}_N\ \mathsf{stat}\,\tau_c\,(v_f\,(\mathsf{dyn}\,\tau_d\,v))$
$\mathsf{stat}\,\tau_d \to \tau_c\,v \quad \mathrm{D}_N\ \mathsf{mon}\,\tau_d \to \tau_c\,v$
$\mathsf{stat}\,\tau_0 \times \tau_1\,v \quad \mathrm{D}_N\ v$
$\mathsf{stat}\,\mathsf{Int}\,v \quad \mathrm{D}_N\ v$
$\mathsf{stat}\,\mathsf{Nat}\,v \quad \mathrm{D}_N\ v$

$\boxed{e \to_{\mathrm{N}}^{*} A}$ reflexive, transitive closure of $\to_{\mathrm{S}_N}$
where $(\to_{\mathrm{S}_N}, \to_{\mathrm{D}_N}) = \mathrm{S}_N \bowtie_E \mathrm{D}_N$

$\boxed{\Gamma \vdash_N e : \tau}$ inherits $\Gamma \vdash e : \tau$:

$$\frac{}{\Gamma \vdash_N \mathsf{mon}\,(\tau_d \to \tau_c)\,v : (\tau_d \to \tau_c)}$$

$\boxed{e\ \mathrm{S}_N\ A}$ extends $e\ \mathrm{S}\ A$:

$(\mathsf{mon}\,\tau_d \to \tau_c\,v_f)\,v\ \mathrm{S}_N\ \mathsf{dyn}\,\tau_c\,(v_f\,(\mathsf{stat}\,\tau_d\,v))$
$\mathsf{dyn}\,\tau_d \to \tau_c\,v \quad \mathrm{S}_N\ \mathsf{mon}\,\tau_d \to \tau_c\,v$
if $v \sim \lambda x.\,e$ or $v \sim \mathsf{mon}\,\tau_d' \to \tau_c'\,v'$
$\mathsf{dyn}\,\tau_0 \times \tau_1\,\langle v_0, v_1\rangle \quad \mathrm{S}_N\ \langle \mathsf{dyn}\,\tau_0\,v_0, \mathsf{dyn}\,\tau_1\,v_1\rangle$
$\mathsf{dyn}\,\mathsf{Int}\,i \quad \mathrm{S}_N\ i$
$\mathsf{dyn}\,\mathsf{Nat}\,i \quad \mathrm{S}_N\ i$
if $i \in \mathbb{N}$
$\mathsf{dyn}\,\tau\,v \quad \mathrm{S}_N\ \mathsf{BoundaryError}$
otherwise

Figure 6: Natural Embedding

example, $(\mathsf{dyn}\ \mathsf{Int}\ \lambda x.\,x)$ has the static type Int but evaluates to a function. Worse yet, well-typed expressions may produce unexpected errors (a category I disaster) or silently compute nonsensical results (a category II disaster).

To illustrate this second kind of danger, recall the classic story of Professor Bessel, who *announced that a complex number was an ordered pair of reals the first of which was nonnegative* [21]. A student might use the type $(\mathsf{Int} \times \mathsf{Nat})$ to model (truncated) Bessel numbers and define a few functions based on the lecture notes. Calling one of these functions with the dynamically-typed value $\langle 1, -4\rangle$ may give a result, but probably not the right one.

Despite the lack of safety, the erasure embedding has found increasingly widespread use. For example, Hack, TypeScript, and Typed Clojure [7] implement this embedding by statically erasing types and re-using the PHP, JavaScript, or Clojure runtime.

3.4 The Natural Embedding

In order to provide some kind of type soundness, an embedding must restrict the dynamically-typed values that can flow in to typed contexts. A natural kind of restriction is to let a value v cross a boundary expecting values of type τ only if v matches the canonical forms of the static type. For base types, this requires a generalized kind of type-tag check. For parameterized types that describe finitely-observable values, this requires one tag check and a recursive check of the value's components.

This inductive checking strategy fails, however, for types that describe values with infinitely many observable behaviors, such as a function or a stream. The classic solution is to use a coinductive strategy and monitor the future behaviors of values [11]. For function types, this means a boundary expecting values of type $(\tau_d \to \tau_c)$ accepts any function value and signals a boundary error if a future application of that value produces a result that does not match the τ_c type. Instead of finding a good reason that the value is typed, the language allows the value as long as there is no evidence that the value is not well-typed.

Figure 6 specifies a natural embedding by extending the multi-language with function monitor values. A monitor $(\mathsf{mon}\,\tau_d \to \tau_c\,v)$ associates a type to a value; new reduction rules ensure that applying the monitor to an argument is the same as applying the underlying value v across two boundary expressions. Statically-typed functions crossing into dynamically-typed code are also wrapped in monitors. Such wrappers check that dynamically-typed arguments match the function's static type.

Monitor values establish a key invariant: every value in statically-typed code is either a well-typed value or a monitor that encapsulates a potentially-dangerous value. This invariant yields a kind of type soundness that is nearly as strong as $\mathbf{L_S}$ soundness.

Theorem ($\mathbf{L_N}$ *type soundness*) If $\vdash e : \tau$ then $\vdash_N e : \tau$ and either:

- $e \to_{\mathrm{N}}^{*} v$ and $\vdash_N v : \tau$
- $e \to_{\mathrm{N}}^{*} E[e'] \to_{\mathrm{D}_N} \mathsf{TagError}$
- $e \to_{\mathrm{N}}^{*} \mathsf{BoundaryError}$
- e diverges

Proof Sketch: By progress and preservation lemmas for the $\Gamma \vdash_N \cdot : \tau$ relation. The lack of type-tag errors in statically-typed code follows from the definition of $\to_{\mathrm{S}_N}$. □

Typed Racket implements the natural embedding by compiling static types to contracts that check dynamically-typed code at run-time [29]. Under the protection of the contracts, Typed Racket may replace certain primitive operations with faster, unsafe versions that are defined for a subset of the Racket value domain [25]. This compilation technique can improve the performance of typed code. However, the overhead of checking the contracts is a significant problem in mixed programs [16, 27].

$\boxed{\text{Language } \mathbf{L_C}}$ inherits $\mathbf{L_N}$:

$v = \ldots \mid \mathsf{mon}\,(\tau \times \tau)\,v$

$\boxed{\Gamma \vdash_C e}$ inherits $\Gamma \vdash_N e$:

$$\frac{}{\Gamma \vdash_C \mathsf{mon}\,(\tau_0 \times \tau_1)\,v}$$

$\boxed{e \;\mathrm{D}_C\; A}$ extends e D A:

$\mathsf{fst}\,(\mathsf{mon}\,\tau_0 \times \tau_1\,v)$	$\mathrm{D}_C\ \mathsf{stat}\,\tau_0\,(\mathsf{fst}\,v)$
$\mathsf{snd}\,(\mathsf{mon}\,\tau_0 \times \tau_1\,v)$	$\mathrm{D}_C\ \mathsf{stat}\,\tau_1\,(\mathsf{snd}\,v)$
$(\mathsf{mon}\,\tau_d \to \tau_c\,v_f)\,v$	$\mathrm{D}_C\ \mathsf{stat}\,\tau_c\,(v_f\ (\mathsf{dyn}\,\tau_d\,v))$
$\mathsf{stat}\,\tau_d \to \tau_c\,v$	$\mathrm{D}_C\ \mathsf{mon}\,\tau_d \to \tau_c\,v$
$\mathsf{stat}\,\tau_0 \times \tau_1\,v$	$\mathrm{D}_C\ \mathsf{mon}\,\tau_0 \times \tau_1\,v$
$\mathsf{stat}\,\mathsf{Int}\,v$	$\mathrm{D}_C\ v$
$\mathsf{stat}\,\mathsf{Nat}\,v$	$\mathrm{D}_C\ v$

$\boxed{e \to_{\mathrm{c}}^{*} A}$ reflexive, transitive closure of $\to_{\mathrm{s}_C}$

where $(\to_{\mathrm{s}_C}, \to_{\mathrm{D}_C}) = \mathrm{s}_C \bowtie_E \mathrm{D}_C$

$\boxed{\Gamma \vdash_C e : \tau}$ inherits $\Gamma \vdash_N e : \tau$:

$$\frac{}{\Gamma \vdash_C \mathsf{mon}\,(\tau_0 \times \tau_1)\,v : (\tau_d \to \tau_c)}$$

$\boxed{e \;\mathrm{s}_C\; A}$ extends e s A:

$\mathsf{fst}\,(\mathsf{mon}\,\tau_0 \times \tau_1\,v)$	$\mathrm{s}_C\ \mathsf{dyn}\,\tau_0\,(\mathsf{fst}\,v)$
$\mathsf{snd}\,(\mathsf{mon}\,\tau_0 \times \tau_1\,v)$	$\mathrm{s}_C\ \mathsf{dyn}\,\tau_1\,(\mathsf{snd}\,v)$
$(\mathsf{mon}\,\tau_d \to \tau_c\,v_f)\,v$	$\mathrm{s}_C\ \mathsf{dyn}\,\tau_c\,(v_f\ (\mathsf{stat}\,\tau_d\,v))$
$\mathsf{dyn}\,\tau_d \to \tau_c\,v$	$\mathrm{s}_C\ \mathsf{mon}\,\tau_d \to \tau_c\,v$
if $v \sim \lambda x.\,e$ or $v \sim \mathsf{mon}\,\tau_d' \to \tau_c'\,v'$	
$\mathsf{dyn}\,\tau_0 \times \tau_1\,v$	$\mathrm{s}_C\ \mathsf{mon}\,\tau_0 \times \tau_1\,v$
if $v \sim \langle v_0, v_1 \rangle$ or $v \sim \mathsf{mon}\,\tau_0' \times \tau_1'\,v'$	
$\mathsf{dyn}\,\mathsf{Int}\,i$	$\mathrm{s}_C\ i$
$\mathsf{dyn}\,\mathsf{Nat}\,i$	$\mathrm{s}_C\ i$
if $i \in \mathbb{N}$	
$\mathsf{dyn}\,\tau\,v$	$\mathrm{s}_C\ \mathsf{BoundaryError}$
otherwise	

Figure 7: Co-Natural Embedding

3.5 Soundness vs. Performance

The erasure embedding promises nothing in the way of type soudness, and lets values freely cross boundary expressions. The natural embedding is ideally type sound (for a language that makes no attempt to connect run-time boundary errors to source-program boundary terms [30]) but imposes a large performance overhead. In the context of Typed Racket, Takikawa etal observed that a straightforward implementation of the natural embedding can slow down a working program by two orders of magnitude.

At a high level, the performance overhead of the natural embedding comes from three sources: checking, indirection, and allocation. By *checking*, we refer to the cost of validating a type-tag and recursively validating the components of a structured value. For example, checking a list structure built from N pair values requires (at least) $2N$ recursive calls. Function monitors add an *indirection* cost. Every call to a monitored function incurs one additional boundary-crossing. If a value repeatedly crosses boundary terms, these type-checking layers can accumulate without bound.[3] Finally, the *allocation* cost of building a monitor value may lead to significant performance overhead.

The following three embeddings address these costs systematically. Consequently, they demonstrate that the erasure and natural embeddings lie on opposite ends of a spectrum between soundness and performance.

3.6 The Co-Natural Embedding

The natural embedding eagerly checks values that cross a type boundary. For most values, this means that a successful boundary-crossing requires a linear-time traversal of the value's components. The exception to this linear-cost rule is the function type. To check that a dynamically-typed value matches a function type, the natural embedding performs a type-tag check and allocates a monitor.

In principle, an embedding can apply the same delayed-checking strategy to values of every parameterized type. This reduces the cost of all boundary terms to at most one type-tag check and one monitor application.

Figure 7 gives the details of this *co-natural* embedding as an extension of the natural embedding. In total, this language $\mathbf{L_C}$ has four kinds of value forms: integers, pairs, functions, function monitors, and pair monitors. The reduction rules define how the projections fst and snd act on pair monitors; in short, they act as projections across a boundary. Finally when a pair value crosses a boundary, it gets wrapping in a checking (or protective) monitor.

From a theoretical standpoint, the change from a natural to a co-natural embedding delays error-checking until just before an expression would reach an undefined state. The co-natural embedding is still type sound in the same sense as the natural embedding:

Theorem ($\mathbf{L_C}$ *type safety*) If $\vdash e : \tau$ then $\vdash_C e : \tau$ and either:

- $e \to_{\mathrm{c}}^{*} v$ and $\vdash_C v : \tau$
- $e \to_{\mathrm{c}}^{*} E[e'] \to_{\mathrm{D}_C} \mathsf{TagError}$
- $e \to_{\mathrm{c}}^{*} \mathsf{BoundaryError}$
- e diverges

Proof Sketch: Similar to the natural embedding. □

There are expressions, however, that reduce to an error in the natural embedding and reduce to a value in the co-natural embedding; for instance $(\mathsf{fst}(\mathsf{dyn}\,\mathsf{Nat} \times \mathsf{Nat}\,\langle 6, -1 \rangle))$.

The switch from eager to delayed run-time checks also affects performance. Instead of checking the contents of a

[3] In a language with a JIT compiler, layers of indirection may also affect inlining decisions.

Language $\mathbf{L_F}$ extends $\mathbf{L_M}$:

$v = \ldots \mid \mathsf{mon}\,(\tau \to \tau)\,(\lambda x.\,e) \mid \mathsf{mon}\,(\tau \to \tau)\,(\lambda(x{:}\tau).\,e) \mid \mathsf{mon}\,(\tau \times \tau)\,\langle v, v\rangle$

$\Gamma \vdash_F e \triangleq \Gamma \vdash_C e$:

$e \;\mathrm{D}_F\; A$ extends $e \;\mathrm{D}\; A$:

$(\mathsf{mon}\,\tau_d \to \tau_c\,(\lambda x.\,e))\;v \quad \mathrm{D}_F\; e[x \leftarrow [\![v]\!]_{\tau_d}]$

$(\mathsf{mon}\,\tau_d \to \tau_c\,(\lambda(x{:}\tau).\,e))\;v \quad \mathrm{D}_F\; \mathsf{stat}\;\tau_c\; e[x \leftarrow [\![v]\!]_{\tau}]$

$\mathsf{fst}\,(\mathsf{mon}\,\tau_0 \times \tau_1\,\langle v_0, v_1\rangle) \quad \mathrm{D}_F\; [\![v_0]\!]_{\tau_0}$

$\mathsf{snd}\,(\mathsf{mon}\,\tau_0 \times \tau_1\,\langle v_0, v_1\rangle) \quad \mathrm{D}_F\; [\![v_1]\!]_{\tau_1}$

$\mathsf{stat}\;\tau\;v \quad \mathrm{D}_F\; [\![v]\!]_{\tau}$

$[\![v]\!]_{\tau} = A$

$[\![v]\!]_{\tau_d \to \tau_c} = \mathsf{mon}\,\tau_d \to \tau_c\,v$
  if $v \sim \lambda x.\,e$ or $v \sim \lambda(x{:}\tau).\,e$

$[\![\mathsf{mon}\,\tau'_d \to \tau'_c\,v']\!]_{\tau_d \to \tau_c} = \mathsf{mon}\,\tau_d \to \tau_c\,v'$

$[\![\langle v_0, v_1\rangle]\!]_{\tau_0 \times \tau_1} = \mathsf{mon}\,\tau_0 \times \tau_1\,\langle v_0, v_1\rangle$

$[\![\mathsf{mon}\,\tau'_0 \times \tau'_1\,v']\!]_{\tau_0 \times \tau_1} = \mathsf{mon}\,\tau_0 \times \tau_1\,v'$

$e \to_{\mathrm{F}}^{*} A$ reflexive, transitive closure of $\to_{\mathrm{S}_F}$
  where $(\to_{\mathrm{S}_F}, \to_{\mathrm{D}_F}) = \mathrm{S}_F \bowtie_E \mathrm{D}_F$

$\Gamma \vdash_F e : \tau \triangleq \Gamma \vdash_C e : \tau$:

$e \;\mathrm{S}_F\; A$ extends $e \;\mathrm{S}\; A$:

$(\mathsf{mon}\,\tau_d \to \tau_c\,(\lambda(x{:}\tau).\,e))\;v \;\mathrm{S}_F\; e[x \leftarrow [\![v]\!]_{\tau}]$

$(\mathsf{mon}\,\tau_d \to \tau_c\,(\lambda x.\,e))\;v \quad \mathrm{S}_F\; \mathsf{dyn}\;\tau_c\; e[x \leftarrow [\![v]\!]_{\tau_d}]$

$\mathsf{fst}\,(\mathsf{mon}\,\tau_0 \times \tau_1\,\langle v_0, v_1\rangle) \quad \mathrm{S}_F\; [\![v_0]\!]_{\tau_0}$

$\mathsf{snd}\,(\mathsf{mon}\,\tau_0 \times \tau_1\,\langle v_0, v_1\rangle) \quad \mathrm{S}_F\; [\![v_1]\!]_{\tau_1}$

$\mathsf{dyn}\;\tau\;v \quad \mathrm{S}_F\; [\![v]\!]_{\tau}$

$[\![i]\!]_{\mathsf{Int}} = i$

$[\![i]\!]_{\mathsf{Nat}} = i$
  if $i \in \mathbb{N}$

$[\![v]\!]_{\tau} = \mathsf{BoundaryError}$
  otherwise

Figure 8: Forgetful Embedding

pair exactly once, at a boundary, the co-natural embedding described in figure 7 performs a check for each application of fst or snd.

3.7 The Forgetful Embedding

The second source of performance overhead in the natural embedding is the indirection cost of monitors. Each time a function value crosses a boundary, it accumulates a new monitor. Pair values in the co-natural embedding suffer the same overhead; a call to fst may factor through an unbounded number of monitor wrappers. To reduce the indirection cost, we need a way to collapse layers of monitors.

A simple, efficient, and type-sound way to reduce the indirection cost is to forget all but the most-recently-applied monitor [14]. When a boundary expecting type τ finds a value of the form $(\mathsf{mon}\,\tau'\,v)$, drop the τ' monitor and return $(\mathsf{mon}\,\tau\,v)$. After all, if a function $(\lambda(x{:}\tau).\,e)$ is well-typed, then the function body e cannot depend on any properties of the old type τ' for soundness.

Figure 8 presents a forgetful, final embedding that co-natural and forgetful monitoring strategies. Intuitively, the only difference between the forgetful language $\mathbf{L_F}$ and the language in figure 7 is that $\mathbf{L_F}$ prevents monitors from stacking up. The details in figure 8 address the fact that monitors in $\mathbf{L_F}$ no longer have a one-to-one correspondence with the type boundaries that a value has crossed. In particular, if the value $(\mathsf{mon}\,\tau\,v)$ is in a dynamically-typed context, then v is *not* necessarily a statically-typed value. We address this potential confusion in two steps. First, when the evaluator applies a function monitor to an argument, it checks whether the call is crossing a type boundary. If so, it interposes a dyn or stat boundary. Second, the boundaries $(\mathsf{stat}\;\tau\;v)$ and $(\mathsf{dyn}\;\tau\;v)$ perform identical checks. The $[\![\cdot]\!]_{\tau}$ meta-function factors out the common work of checking a value and dropping any existing monitor.

The language $\mathbf{L_F}$ satisfies the same notion of soundness as the co-natural $\mathbf{L_C}$ and the natural $\mathbf{L_N}$:

Theorem ($\mathbf{L_F}$ *type soundness*) If $\vdash e : \tau$ then $\vdash_F e : \tau$ and either:

- $e \to_{\mathrm{F}}^{*} v$ and $\vdash_F v : \tau$
- $e \to_{\mathrm{F}}^{*} E[e'] \to_{\mathrm{D}_F} \mathsf{TagError}$
- $e \to_{\mathrm{F}}^{*} \mathsf{BoundaryError}$
- e diverges

Proof Sketch: By progress and preservation of the $\vdash_F \cdot : \tau$ relation. The key invariant is that if x is a variable of type τ, then the forgetful semantics ensures that the value substituted for x has the expected type. This value may be different from the value substituted by the natural semantics, but that distinction is not important for proving type soundness. □

The forgetful embedding performs just enough run-time type checking to ensure that statically-typed code does not reach an undefined state, nothing more.

3.8 The Locally-Defensive Embedding

The final source of performance overhead in the natural embedding is the cost of allocating monitor values. To remove

Language $\mathbf{L_K}$ inherits $\mathbf{L_M}$:

$$\begin{array}{lcl} e & = & \ldots \mid \mathsf{dyn\ Any}\ e \mid \mathsf{stat\ Any}\ e \mid \mathsf{check}\ K\ e \\ K & = & \mathsf{Nat} \mid \mathsf{Int} \mid \mathsf{Pair} \mid \mathsf{Fun} \mid \mathsf{Any} \\ E & = & \ldots \mid \mathsf{check}\ K\ E \end{array}$$

$\Gamma \vdash_K e$ inherits $\Gamma \vdash_D e$:

$$\frac{(x:\tau), \Gamma \vdash_K e : \mathsf{Any}}{\Gamma \vdash_K \lambda(x:\tau).\, e} \qquad \frac{(x:\tau) \in \Gamma}{\Gamma \vdash_K x} \qquad \frac{\Gamma \vdash_K e : \lfloor \tau \rfloor}{\Gamma \vdash_K \mathsf{stat}\ \tau\ e}$$

$$\frac{\Gamma \vdash_K e : \mathsf{Any}}{\Gamma \vdash_K \mathsf{stat\ Any}\ e}$$

$e\ \mathrm{D}_K\ A$ extends $e\ \mathrm{D}\ A$:

$$\begin{array}{l} (\lambda(x:\tau).\, e)\ v\ \mathrm{D}_K\ \mathsf{stat\ Any}\ (e[x \leftarrow [\![v]\!]_K]) \\ \quad \text{where } K = \lfloor \tau \rfloor \\ \mathsf{stat}\ \tau\ v \quad \mathrm{D}_K\ [\![v]\!]_K \\ \quad \text{where } K = \lfloor \tau \rfloor \\ \mathsf{stat\ Any}\ v \quad \mathrm{D}_K\ v \end{array}$$

$e\ \mathrm{S}_K\ A$ extends $e\ \mathrm{S}\ A$:

$$\begin{array}{l} (\lambda x.\, e)\ v \quad \mathrm{S}_K\ \mathsf{dyn\ Any}\ e[x \leftarrow v] \\ \mathsf{dyn}\ \tau\ v \quad \mathrm{S}_K\ [\![v]\!]_K \\ \quad \text{where } K = \lfloor \tau \rfloor \\ \mathsf{dyn\ Any}\ v\ \mathrm{S}_K\ v \\ \mathsf{check}\ K\ v\ \mathrm{S}_K\ [\![v]\!]_K \end{array}$$

$[\![v]\!]_K = A$

$$\begin{array}{ll} [\![v]\!]_K = v & \text{if } \vdash_K v : K \\ [\![v]\!]_K = \mathsf{BoundaryError} & \text{otherwise} \end{array}$$

$\lfloor \tau \rfloor = \kappa$

$$\begin{array}{lcl} \lfloor \tau_d \to \tau_c \rfloor & = & \mathsf{Fun} \\ \lfloor \tau_0 \times \tau_1 \rfloor & = & \mathsf{Pair} \\ \lfloor \mathsf{Nat} \rfloor & = & \mathsf{Nat} \\ \lfloor \mathsf{Int} \rfloor & = & \mathsf{Int} \end{array}$$

$K <: K$

$$\frac{}{\mathsf{Nat} <: \mathsf{Int}} \qquad \frac{}{K <: \mathsf{Any}}$$

$\Gamma \vdash_K e : K$ (selected rules):

$$\frac{\Gamma \vdash_K e_0 : \mathsf{Any} \quad \Gamma \vdash_K e_1 : \mathsf{Any}}{\Gamma \vdash_K \langle e_0, e_1 \rangle : \mathsf{Pair}} \qquad \frac{x, \Gamma \vdash_K e}{\Gamma \vdash_K \lambda x.\, e : \mathsf{Fun}}$$

$$\frac{(x:\tau), \Gamma \vdash_K e : \mathsf{Any}}{\Gamma \vdash_K \lambda(x:\tau).\, e : \mathsf{Fun}} \qquad \frac{x \in \Gamma}{\Gamma \vdash_K x : \mathsf{Any}} \qquad \frac{(x:\tau) \in \Gamma \quad \lfloor \tau \rfloor = K}{\Gamma \vdash_K x : K}$$

$$\frac{\Gamma \vdash_K e_0 : \mathsf{Fun} \quad \Gamma \vdash_K e_1 : \mathsf{Any}}{\Gamma \vdash_K e_0\ e_1 : \mathsf{Any}} \qquad \frac{\Gamma \vdash_K e : \mathsf{Pair}}{\Gamma \vdash_K \mathsf{fst}\ e : \mathsf{Any}} \qquad \frac{\Gamma \vdash_K e : \mathsf{Pair}}{\Gamma \vdash_K \mathsf{snd}\ e : \mathsf{Any}}$$

$$\frac{\Gamma \vdash_K e \quad \lfloor \tau \rfloor = K}{\Gamma \vdash_K \mathsf{dyn}\ \tau\ e : K} \qquad \frac{\Gamma \vdash_K e}{\Gamma \vdash_K \mathsf{dyn\ Any}\ e : \mathsf{Any}}$$

$$\frac{\Gamma \vdash_K e : K'}{\Gamma \vdash_K \mathsf{check}\ K\ e : K} \qquad \frac{\Gamma \vdash_K e : K' \quad K' <: K}{\Gamma \vdash_K e : K}$$

$e \to_{\mathrm{K}}^{*} A$ reflexive, transitive closure of $\to_{\mathrm{S}_K}$
where $(\to_{\mathrm{S}_K}, \to_{\mathrm{D}_K}) = \mathrm{S}_K \bowtie_E \mathrm{D}_K$

Figure 9: Locally-Defensive Embedding

this cost, we must replace the run-time analysis that monitors implement with a static analysis that predicts where to insert soundness-enforcing run-time checks.

For a typical notion of type soundness in a language with first-class functions, predicting such checks is impossible without a whole-program analysis. The key insight is to pick a sufficiently weak notion of soundness [32]. Our contribution is to demonstrate that this type-tag soundness arises systematically from the co-natural and forgetful embeddings.

3.8.1 Monitor-Free Semantics

The co-natural embedding decomposes deep checks at a boundary expression to one shallow check at the boundary and a set of shallow checks, one for each time the value is observed. The forgetful embedding ignores the history of a value; local type annotations completely determine the type checks in a context. Combining these two embeddings yields another semantics with local, defensive type-tag checks. In $\mathbf{L_F}$, these defensive checks occur in exactly three kinds of expressions: (1) at boundary terms, (2) at function call and return, and (3) at calls to the projections fst and snd.

We can statically determine whether an expression *might* require a run-time check with a typing system that assumes any structured value can produce any kind of value. Function applications $(e_0\ e_1)$ and projections $(\mathsf{fst}\ e)$ in this system need a type that represents any kind of value. In order to use these expressions in a context that requires a certain kind of value, for example $(\mathsf{sum}\ 2\ [\,])$, we add a form $(\mathsf{check}\ K\ e)$ to internalize the notion of a type-tag check.

Figure 9 defines a typing system $\Gamma \vdash_K e : K$ that makes these ideas precise. To begin, type-tags K represent integers, natural numbers, pairs, functions, and unknown values. The meta-function $\lfloor \cdot \rfloor$ relates a type τ to a type-tag, and the subtyping relation $K <: K'$ states when values of tag K can safely be given to a context expecting values of a different tag. As for the typing system: (1) the rules for value constructors conclude with a non-trivial tag; (2) the rules for elimination forms require a non-trivial tag and conclude with the Any tag; and (3) the rules for dyn and check conclude a non-trivial tag that is justified with a run-time check.

Figure 9 additionally defines a semantics for well-tagged expressions. Crucially, dynamically-typed arguments to typed

$$\boxed{\Gamma \vdash_S e : \tau \rightsquigarrow e}$$

$$\frac{\Gamma \vdash e_0 : \tau_d \to \tau_c \rightsquigarrow e_0' \quad \Gamma \vdash e_1 : \tau_d \rightsquigarrow e_1' \quad \lfloor \tau_c \rfloor = K}{\Gamma \vdash e_0\ e_1 : \tau_c \rightsquigarrow \mathsf{check}\ K\ e_0'\ e_1'}$$

$$\frac{\Gamma \vdash e : \tau_0 \times \tau_1 \rightsquigarrow e' \quad \lfloor \tau_0 \rfloor = K}{\Gamma \vdash \mathsf{fst}\ e : \tau_0 \rightsquigarrow \mathsf{check}\ K\ \mathsf{fst}\ e'}$$

$$\frac{\Gamma \vdash e : \tau_0 \times \tau_1 \rightsquigarrow e' \quad \lfloor \tau_1 \rfloor = K}{\Gamma \vdash \mathsf{snd}\ e : \tau_1 \rightsquigarrow \mathsf{check}\ K\ \mathsf{snd}\ e'}$$

Figure 10: Completion (selected rules)

functions get tag-checked by the $[\![\cdot]\!]_K$ meta-function. Also important for proving that type-tag errors only occur in dynamically-typed code is the addition of the "dummy" boundary expressions (dyn Any e) and (stat Any e). These expressions are a technical device to quarantine function bodies.

3.8.2 Check Completion

Unlike our previous languages, there is a gap between static typing for the multi-language $\mathbf{L_M}$ and the language $\mathbf{L_K}$ in figure 9. If an expression e has the static type τ and the type-tag of τ is K, it is not necessarily true that e is well-tagged via the $\vdash_K$ relation.

We bridge this gap with a *completion* [17] function. Informally, a completion function $\rightsquigarrow$ takes a well-typed term $\vdash e : \tau$ and adds check forms to enforce the type-checker's assumptions against dynamically-typed pairs and functions. Such a function is correct if it maps well-typed expressions to semantically-equivalent well-tagged expressions.

For $\mathbf{L_K}$, the completion function we use inserts checks to the three forms shown in figure 10 and otherwise folds over expressions. We leave as open the question of how to define a completion function that generates the minimum number of check expressions.[4]

3.8.3 Type-Tag Soundness

We state soundness for $\mathbf{L_K}$ in terms of the static typing judgment of the mixed language and the semantics of $\mathbf{L_K}$.

Theorem ($\mathbf{L_K}$ *type-tag soundness*) If $\vdash e : \tau$ and $\lfloor \tau \rfloor = K$, then $\vdash e : \tau \rightsquigarrow e^+$ and $\vdash_K e^+ : K$ and either:

- $e^+ \to_{\mathrm{K}}^* v$ and $\vdash_K v : K$
- $e^+ \to_{\mathrm{K}}^* E[e'] \to_{\mathrm{D}_K} \mathsf{TagError}$
- $e^+ \to_{\mathrm{K}}^* \mathsf{BoundaryError}$
- e^+ diverges

Proof Sketch: The completion function simply adds checks around every expression with type-tag Any to enforce the expression's static type. Soundness follows from progress and preservation lemmas for the $\vdash_K \cdot : K$ relation. □

[4] Henglein [17] defines a rewriting system that is provably optimal, but possibly non-terminating.

4 From Models to Implementations

The models from the preceding section do not address the full range of types found in practical languages. Here we sketch how to address these limitations.

4.1 Compiling to a Host Language

The models employ a small-step operational semantics for an expression language. Indeed, the type-sound ones (natural, co-natural, forgetful, and locally-defensive) use two mutually-recursive reduction relations. In practice, though, a migratory typing system for a language $\mathbf{L_D}$ compiles statically-typed code to this host language, which raises two questions.

The first question is how to represent the static types that the models use in monitor values and function applications. A suitable compiled representation for $(\mathsf{mon}\ \tau_d \to \tau_c\ v)$ is $(\mathsf{mon}\ \langle e_d, e_c \rangle\ v)$ where e_d is a host-language function that checks whether a value matches the domain type. In the forgetful variant, the domain of $(\lambda(x:\tau).\ e)$ can replace the domain type in its enclosing monitor. In the locally-defensive variant, $(\lambda(x:\tau).\ e)$ compiles to a function that checks the actual value of x against the type τ before executing the function body.

The second question is whether it is sound to use the $\mathbf{L_D}$ reduction relation on statically-typed terms. Indeed, all of our models do not need separate reduction relations other than for the soundness proofs in the preceding section. The reductions differ in two minor aspects: how they interpose boundary terms and how many run-time checks they perform. As for the boundaries, they become irrelevant in an implementation because the set of values are the same.As for the run-time checks, the static reduction can skip checks that the dynamic reduction must perform, i.e., it is safe to use the more conservative, dynamically-typed reduction relation.

4.2 Tags for Additional Types

The literature on migratory typing describes methods for implementing a variety of types, including untagged union types [29] and structural class types [26]. Those techniques apply to the co-natural and forgetful variants, though only if we ignore precise blame information for dynamically discovered type violations [9].

Techniques for implementing the locally-defensive variant are less well-known, so we describe a few here. To support *types for mutable data*, it suffices to tag-check every read from a mutable data structure. If all reads are checked, then writes to a mutable value do not require a tag check.

To support *structural class types* and functions with *optional and keyword arguments*, a language designer has two choices. One choice is to simply check that the incoming value is a class or procedure. A second is to use reflective

operations (if the language supports them) to count the methods and arity of the incoming value. In our experience, the latter does not add significant overhead.

To support *untagged union types*, the language of tags K requires a matching tag combinator. Let or be this constructor; the tag for a union type $(\cup\ \tau_0\ \tau_1)$ is then $(\text{or}\ K_0\ K_1)$ where K_i is the tag of type τ_i.

To support *recursive types* of the form $(\mu\,\alpha.\,\tau)$, a suitable type-tag is the tag of $(\tau[\alpha \leftarrow \mu\,\alpha.\,\tau])$. This definition is well-founded provided the type variable α appears only as the parameter to a *guarded* type. A parameterized type is guarded if its type-tag does not depend on its argument types.

To support *universal types* of the form $\forall\,\alpha.\,\tau$, we use the tag $\lfloor\tau\rfloor$ and define $\lfloor\alpha\rfloor = \text{Any}$. Intuitively, there is no need to tag-check the use of a type variable because type variables have no elimination forms.

5 Performance Evaluation

Based on the models, the natural, locally-defensive, and erased embeddings seem to occupy three distinct points on a spectrum between soundness and performance. To measure how these embeddings stack up as competing implementation strategies for the same host language and typing system, we have implemented a locally-defensive embedding as an extension of Typed Racket. Since Typed Racket implements a natural embedding, this allows us to compare the three approaches:

- the natural embedding, via Typed Racket;
- the locally-defensive embedding, via the prototype;
- and the erasure embedding, via Racket.

By contrast, we view the co-natural and forgetful embeddings as theoretical artifacts. An implementation of the natural embedding might benefit from more co-natural laziness, and an implementation of the locally-defensive embedding might benefit from careful use of monitors, but we leave these questions for future work.

5.1 Implementation Overview

Our implementation of the locally-defensive embedding exists as a fork of Typed Racket v6.10.1 called Tagged Racket. Tagged Racket inherits the syntax and static type checker of Typed Racket. Tagged Racket extends Typed Racket with a type-to-tag compiler and a completion function for type-annotated programs.

The type-to-tag function compiles a representation of a static type to a Racket predicate that checks whether a value matches the type. The completion function traverses a well-typed program and inserts two kinds of checks. It adds an `assert` statement to every statically-typed function to defend the function body against dynamically-typed arguments. It wraps any destructor calls with a similar assert to confirm that the destructor returns a result with the expected tag.

The implementation is designed to support a functional subset of Racket. It does not support Typed Racket's class and object system [26].

5.2 Evaluation Method

The promise of migratory typing is that programmers can freely mix statically-typed and dynamically-typed code. A performance evaluation of a migratory typing system must therefore give programmers a sense of the performance they can expect as they add static typing to a program [16, 27].

To meet this goal, we measure the performance of all configurations of statically-typed and dynamically-typed code in a suite of Racket programs. Since Typed Racket (and our prototype) allows module-level type boundaries, this means that a Racket program with N modules has 2^N configurations.

To turn this raw data into a more direct message, we use the notion of a D-deliverable configuration from Takikawa et al. [27] A configuration is D-deliverable if its performance overhead relative to a fixed baseline configuration is at most D. The baseline we use is the performance of Racket. We refer to this as the untyped configuration. Its performance corresponds to an erasure embedding.

Remark The premise of the D-deliverable measure is that programmers have a fixed performance requirement. Certain applications may have strict performance requirements and can only tolerate a 10% overhead, corresponding to $D = 1.1x$. Others may accept overhead as high as $D = 10x$. No matter the requirement, any programmer can instantiate D and check whether the proportion of D-deliverable configurations is high enough to enable an incremental transition to a typed codebase. *End*

The programs we use are adapted from the functional benchmarks of Takikawa et al. [27]. See the appendix for details and origins of each benchmark.

To measure performance, we ran each configuration 8 times using Racket v6.10.1 on an unloaded Linux machine with two physical AMD Opteron 6376 processors[5] and 128GB RAM. The CPU cores on each processor were all configured to run at 2.30 GHz using the performance CPU governor. We did not collect measurements in parallel.

5.3 Results

The plots in figure 11 report the performance of Typed Racket (the natural embedding) and of Tagged Racket (the locally-defensive embedding) as two histograms. In the plot for a benchmark B, the data for Typed Racket is a blue line and the data for Tagged Racket is an orange line. A point (X, Y) on the line for Typed Racket says that Y% of all Typed Racket configurations for the benchmark B run at most X times slower than Racket running the same code with all types erased. The line for Tagged Racket is analogous. Note that

[5]The Opteron is a NUMA architecture.

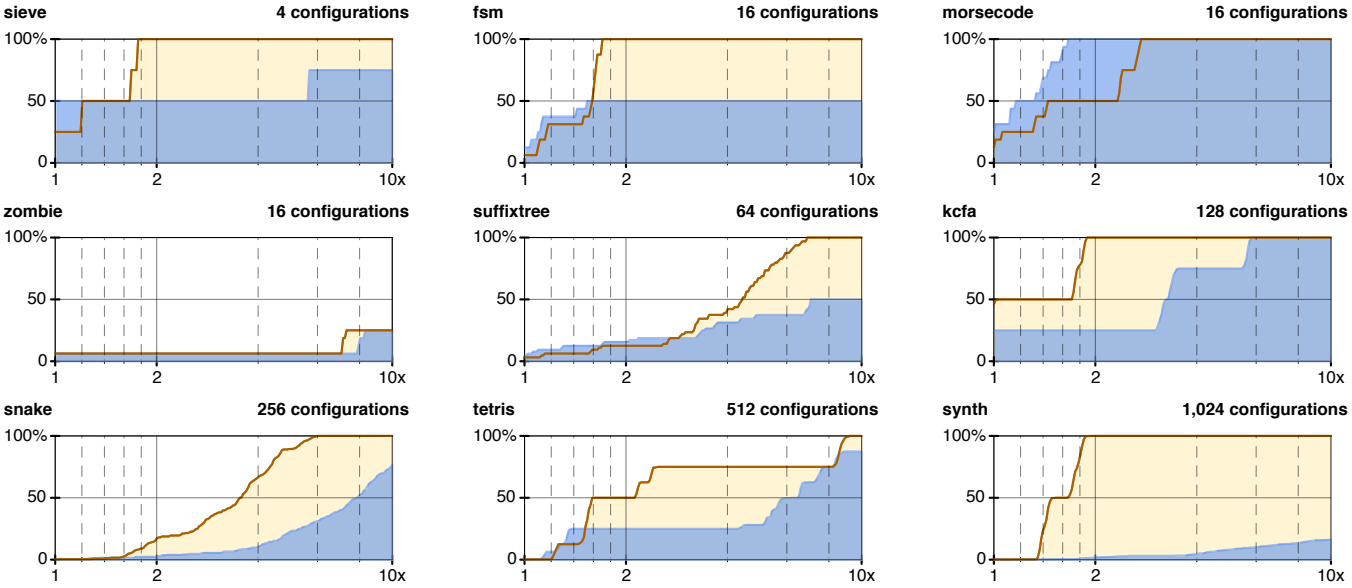

Figure 11: Tagged Racket (orange) vs. Typed Racket (blue)

the x-axis is log scaled; vertical tick marks appear at 1.2x, 1.4x, 1.6x, 1.8x, 4x, 6x, and 8x overhead.

The data confirms that Tagged Racket is significantly more performant than Typed Racket; see the plots for `sieve`, `fsm`, `suffixtree`, `kcfa`, `snake`, `tetris`, and `synth`. The improvement is most dramatic for `synth`, in which the worst-case performance ovehead drops from 39x to 2x, mostly because Typed Racket `synth` spends a large amount of time eagerly traversing data structures and monitoring their components.

The `zombie` benchmark shows only a minor improvement; few configurations are 10-deliverable in either Typed Racket or Tagged Racket. We remark, however, that the worst-case overhead in Typed Racket for `zombie` is 543x whereas the worst-case for Tagged Racket is far lower, at 27x.

The `morsecode` benchmark is an anomaly. Using Tagged Racket increases its worst-case overhead from 1.7x to 2.7x. This degredation occurs because the pervasive type-tag checks of Tagged Racket introduce more overhead than the boundary checks inserted by Typed Racket.

More broadly, the overhead in `morsecode` speaks to a general trend: as the amount of statically-typed code increases, the performance overhead of Tagged Racket increases linearly. See the Appendix for details.

5.4 Threats to Validity

The performance of our Tagged Racket prototype is an order-of-magnitude improvement over Typed Racket. We believe this high-level conclusion is valid; however, the exact performance of a full implementation is likely to vary from our prototype implementation.

On one hand, the prototype is likely to be faster than a complete implementation because it makes little effort to provide useful error messages. When a tag check fails, the prototype simply directs programmers to the source code associated with the tag check using information available to the completion function. Improving these error messages with information about the source of an ill-tagged value is likely to degrade performance.

Similarly, the prototype avoids using Racket's contract system to implement type-tag checks. Contracts are a useful tool for defining predicates that give well-structured error messages, but they can add prohibitive overheads. Perhaps the implementation of contracts could be improved; perhaps they are just the wrong tool for implementing frequently-executed assert statements.

On the other hand, the performance of the prototype could be improved in two obvious ways. First, Tagged Racket does not take advantage of the Typed Racket optimizer to remove type-tag checks from primitive operations. Second, the prototype could use type-based static analysis to detect redundant type-tag checks.

Three other threats are worth noting. First, Tagged Racket does not support Racket's object-oriented features; programs using such features might not improve as drastically as the functional benchmarks we measure. Second, our benchmarks are relatively small; the largest is 10 modules and 800 lines (see appendix for full details). Third, ascribing different types to the same program can affect its performance; for example the tag check for an integer is less expensive than the tag check for a natural number or some other union type. Nevertheless we consider our results representative.

6 Related Work

Gradual Typing In the broad sense, the term gradual typing [23] describes any research that combines static and dynamic typing. In the more precise sense defined by Siek et al. [24], a gradual typing system includes: (1) a dynamic type that may be implicitly cast to any other type; (2) a relation between types that are equal up to occurrences of the dynamic type; and (3) a proof that replacing any static type with the dynamic type can only affect the semantics of a term by removing a boundary error.

Migratory Typing Tobin-Hochstadt and Felleisen introduced the idea of migratory typing with Typed Racket [28]. This led to a series of works on designing types to accomodate the idioms of dynamically-typed Racket; see Tobin-Hochstadt et al. [30] for an overview. Other migratory typing systems target JavaScript [5], Python [31], and Smalltalk [1].

Gradual Typing Performance Takikawa et al. [27] introduce a method for systematically evaluating the performance of a gradual typing system. They apply the method to Typed Racket and find that its performance is a serious problem. Bauman et al. [4] apply the method to a tracing JIT back-end for a subset of Typed Racket and report order-of-magnitude improvements without sacrificing soundness, performance, or error messages.

Greenman and Migeed [15] apply the Takikawa evaluation method to Reticulated and find that the overhead of type-tag soundness is within 10x on their benchmarks. Muehlboeck and Tate [19] evaluate the performance of an object calculus and report that natural-embedding gradual typing can yield an efficient implementation if the language of types is restricted to a finite set of (unparameterized) class names.

Type-Tag Soundness Vitousek et al. [32] present a compiler from a statically-typed source language to a Python-like target language and prove a type-tag soundness theorem. Using the ideas from the calculus, they implement a locally-defensive embedding of Reticulated (a statically typed variant of Python) into Python 3. Their so-called transient semantics is the motivation for our work; we began by trying to implement a variant of their compiler for Typed Racket, but needed a deeper understanding of why the compiler was correct, what design choices were essential, and how the idea of rewriting statically-typed code related to other multi-language embeddings.

The idea of rewriting an expression to add explicit safety checks goes back at least to Henglein [17], from whom we adopt the name *completion*.

Final Algebra Semantics The co-natural embedding is directly inspired by Findler and Felleisen's technique for higher-order contracts [11] and prior work on lazy contract checking [12]. We conjecture that this embedding could be viewed as a *final algebra semantics* [33].

Spectrum of Type Soundness Two related works led us towards the idea of a spectrum of type soundness. *Like types* are static type annotations that are erased before run-time [6, 22]. Programmers can switch between like types and normal types to exchange soundness for performance without sacrificing any static type checking.

The *progressive types* vision paper describes a type system in which programmers can decide whether certain errors are caught statically or dynamically [20]. This offers a choice between (1) statically proving an expression is universally correct, and (2) letting the run-time dynamically check whether the code is safe in practice.

7 Finding Balance

The paper contributes two major results. First, the idea of viewing migratory typing as a multi-language problem demonstrates that a language with a migratory type system may satisfy at least five different type-soundness conditions, depending on which kind of "foreign-function interface" we choose. Each soundness condition has different implications for how a developer can reason about the code, especially when it comes to diagnosing the cause of a run-time error.

- Running a Typed Racket program as a Racket program (via erasure) gives a developer no clue as to what triggers an error; the type information in the code does *not* reduce the search space. Indeed, a violation of the types in the source code may go unnoticed.
- Running a Typed Racket program in Tagged Racket is guaranteed to reveal a violation of types *eventually* if it affects the execution. The delayed checking schema may completely obscure the source of the error, however.
- Running a Typed Racket program with the full contract checks uncovers a violation of type annotations as soon as there is a witness.

Second, our measurements are the first "apples-to-apples" comparison, and they confirm that a locally defensive semantics reduces the cost of a migratory run-time checking by at least an order of magnitude. We conjecture that additional improvements on the checking scheme and the Tagged Racket implementation may bring the performance within a reasonable overhead factor for every mixed-typed configuration.

The question for researchers is thus what developers really want from a migratory type system. More specifically, we need to ask how much performance they are willing to sacrifice for how much soundness. We expect that where there are five flavors of soundness, there might be many more, and additional work may just find one that properly balances the need for performance with the need for guarantees.

Bibliography

[1] Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for Smalltalk. *Science of Computer Programming* 96(1), pp. 52–69, 2013.

[2] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland Publishing Company, 1981.

[3] Edd Barrett, Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Fine-grained Language Composition: A Case Study. In *Proc. European Conference on Object-Oriented Programming*, pp. 3:1–3:27, 2016.

[4] Spenser Bauman, Sam Tobin-Hochstadt, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. Sound Gradual Typing: Only Mostly Dead. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2017.

[5] Gavin Bierman, Martin Abadi, and Mads Torgersen. Understanding TypeScript. In *Proc. European Conference on Object-Oriented Programming*, pp. 257–281, 2014.

[6] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: Robust, Concurrent, Extensible Scripting on the JVM. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 117–136, 2009.

[7] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical Optional Types for Clojure. In *Proc. European Symposium on Programming*, pp. 68–94, 2016.

[8] Robert L. Constable. Mathematics as Programming. In *Proc. Workshop on Logic of Programs*, pp. 116–128, 1983.

[9] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete Monitors for Behavioral Contracts. In *Proc. European Symposium on Programming*, pp. 214–233, 2012.

[10] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2010.

[11] Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *Proc. ACM International Conference on Functional Programming*, pp. 48–59, 2002.

[12] Robert Bruce Findler, Shu-yu Guo, and Anne Rogers. Lazy Contract Checking for Immutable Data Structures. In *Proc. International Symposium Functional and Logic Programming*, pp. 111–128, 2007.

[13] Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-Grained Interoperability Through Mirrors and Contracts. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 231–245, 2005.

[14] Michael Greenberg. Space-Efficient Manifest Contracts. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 181–194, 2015.

[15] Ben Greenman and Zeina Migeed. On the Cost of Type-Tag Soundness. To appear in PEPM, 2018.

[16] Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. How to Evaluate the Performance of Gradual Type Systems. Submitted for publication, 2017.

[17] Fritz Henglein. Dynamic Typing: Syntax and Proof Theory. *Science of Computer Programming* 22(3), pp. 197–230, 1994.

[18] Jacob Matthews and Robert Bruce Findler. Operational Semantics for Multi-Language Programs. *Transactions on Programming Languages and Systems* 31(3), pp. 12:1–12:44, 2009.

[19] Fabian Muehlboeck and Ross Tate. Sound Gradual Typing is Nominally Alive and Well. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2017.

[20] Joe Gibbs Politz, Hannah Quay-de la Vallee, and Shriram Krishnamurthi. Progressive Types. In *Proc. ACM Symposium on New ideas, new paradigms, and reflections on programming and software*, pp. 55–66, 2012.

[21] John C. Reynolds. Types, Abstraction, and Parametric Polymorphism. In *Proc. Information Processing*, 1983.

[22] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete Types for TypeScript. In *Proc. European Conference on Object-Oriented Programming*, pp. 76–100, 2015.

[23] Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Proc. Scheme and Functional Programming Workshop*, 2006.

[24] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined Criteria for Gradual Typing. In *Proc. Summit oN Advances in Programming Languages*, pp. 274–293, 2015.

[25] Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. Typing the Numeric Tower. In *Proc. Symposium on Practical Aspects of Declarative Languages*, pp. 289–303, 2012.

[26] Asumu Takikawa, Daniel Feltey, Earl Dean, Robert Bruce Findler, Matthew Flatt, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards practical gradual typing. In *Proc. European Conference on Object-Oriented Programming*, pp. 4–27, 2015.

[27] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is Sound Gradual Typing Dead? In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 456–468, 2016.

[28] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage Migration: from Scripts to Programs. In *Proc. Dynamic Languages Symposium*, pp. 964–974, 2006.

[29] Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 395–406, 2008.

[30] Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory Typing: Ten years later. In *Proc. Summit oN Advances in Programming Languages*, 2017.

[31] Michael M. Vitousek, Andrew Kent, Jeremy G. Siek, and Jim Baker. Design and Evaluation of Gradual Typing for Python. In *Proc. Dynamic Languages Symposium*, pp. 45–56, 2014.

[32] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big Types in Little Runtime: Open-World Soundness and Collaborative Blame for Gradual Type Systems. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 762–774, 2017.

[33] Mitchell Wand. Final Algebra Semantics and Data Type Extension. *Journal of Computer and System Sciences* 19, pp. 27–44, 1979.

[34] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, pp. 38–94, 1994.