A Spectrum of Type Soundness and Performance

BEN GREENMAN, PLT @ Northeastern University, USA

MATTHIAS FELLEISEN, PLT @ Northeastern University, USA

The literature on gradual typing presents three fundamentally different ways of thinking about the integrity of programs that combine statically typed and dynamically typed code. This paper presents a uniform semantic framework that explains all three approaches, illustrates how each approach affects a developer's work, and adds a systematic performance comparison for a single implementation platform.

CCS Concepts: • **Software and its engineering** → **Semantics**; Software evolution;

Additional Key Words and Phrases: migratory typing, type soundness, performance evaluation, D-deliverable

1 THREE FLAVORS OF MIGRATORY TYPING

For the past two decades, many programmers have built systems in dynamically typed programming languages. Regardless of why they make this choice, they eventually discover that they wish their code base came with some types. To accommodate the *migration* of a large code base from an untyped language to a typed one, researchers have created migratory typing systems [82]. In essence, a migratory typing system comes with the same expression and statement syntax as the underlying untyped language but allows the addition of type annotations. While all such systems use the annotations for static analysis [12, 13, 15, 16, 18, 27, 44, 45, 58, 59, 62, 82, 84, 89], it remains unclear what these type annotations *mean* for the behavior of a mixed-typed program.

Over the years, three approaches have emerged for interpreting types in a mixed-typed setting. Each approach generalizes type soundness from one language to a pair of related languages. The first approach is to *enforce types eagerly* at the boundaries between statically and dynamically typed code, which leads to a generalized form of traditional type soundness [66, 79]. Eager enforcement of higher-order types prevents dynamically typed code from sending (type) invalid arguments to a typed function or returning invalid results to a typed context from untyped functions. But, it may impose a significant run-time cost [33, 76]. A second approach is to *erase the types* and rely on the soundness of the underlying dynamically typed language [12]. While this lack of any dynamic enforcement is free of run-time overhead, it takes a "garbage in, garbage out" approach toward interactions between the statically typed and dynamically typed parts of a mixed-typed program. Finally, a third approach is to compromise between those two extremes and to *check type constructors* in a way that protects typed code against first-order errors [84].

The existence of three approaches raises two scientific question concerning a proper comparison:

Authors' addresses: Ben Greenman, PLT @ Northeastern University, USA, benjaminlgreenman@gmail.com; Matthias Felleisen, PLT @ Northeastern University, USA, matthias@ccs.neu.edu.

- *How do the logical implications of the three approaches compare?*
  Publications on implementations of migratory typing often prove a "type soundness" (or "type safety") theorem without formally discussing how soundness for the pair of languages differs from soundness for a single language [58, 84].
  To answer this question, this paper explains the three approaches in a systematic manner within one semantic framework. For the same source syntax and type system, it formulates the three approaches as three different semantics and states three precise soundness theorems. It also illustrates the consequences of each theorem for developers.
- *How do the three approaches compare with respect to performance?*
  Researchers in this area have only recently begun to study the performance of implementations systematically [11, 33, 51, 76]. Previous attempts to compare approaches make claims about *different programming languages* using mostly-unrelated benchmarks [32, 51, 84].
  To answer this question properly, the paper measures the same benchmarks in three implementations of the same syntax and type system, based on the common theoretical framework. While our results confirm the published conjectures to some degree, we consider it imperative for the future of this research area to put such comparisons on solid ground.

2　SYNTAX, TYPES, AND SEMANTICS

The three approaches to migratory typing can be understood as three multi-language embeddings in the style of Matthews and Findler [46]. Each approach uses a different strategy to enforce static types at the boundaries between typed and untyped code: eagerly enforcing types corresponds to a *higher-order* embedding; ignoring types corresponds to an *erasure* embedding; and enforcing type constructors corresponds to a *first-order* embedding.

　　This section first introduces the surface syntax and typing system (section 2.1). It then defines three models, states their soundness theorems (sections 2.3, 2.4, and 2.5), and concludes with a discussion on scaling the models to a practical implementation (section 2.6). Each model builds upon a common semantic framework (section 2.2) to keep the technical presentation focused on their differences. For unabridged definitions, we refer the reader to the supplementary material [31].

2.1　Common Syntactic Notions

A migratory typing system extends a dynamically-typed host language with syntax for type annotations. The type checker for the extended language must be able to validate mixed-typed programs, and the semantics must define a type-directed protocol for transporting values across the boundaries between typed and untyped regions of code.

　　In a full-fledged language, all kinds of values may cross a type boundary at run-time: values of base type (numbers, strings, booleans), values of algebraic type (pairs, finite lists, immutable sets), and values of higher type (functions, mutable references, infinite lists). As representative examples, the surface language in figure 1 includes integers, pairs, and functions, and three corresponding types. The fourth type, Nat, is a subset of the type of integers and is included because set-based reasoning is common in dynamically-typed programs [3, 81, 82].

　　An expression in the surface language may be dynamically typed (e_D) or statically typed (e_S). Each grammar includes a *boundary term* for embedding an expression of the other grammar. The expression (dyn τ e_D) embeds a dynamically-typed subexpression in a statically-typed context, and the expression (stat τ e_S) embeds a statically-typed subexpression in a dynamically-typed context.

　　The last two equations in figure 1 specify the names of primitive operations (op^1, op^2). The primitives represent low-level procedures that manipulate the machine-level representation of values (i.e., bitstrings).

Surface Syntax

$$e_D = x \mid \lambda x.\, e_D \qquad \mid i \mid \langle e_D, e_D \rangle \mid e_D\, e_D \mid op^1\, e_D \mid op^2\, e_D\, e_D \mid \text{stat}\ \tau\ e_S$$
$$e_S = x \mid \lambda(x{:}\tau).\, e_S \mid i \mid \langle e_S, e_S \rangle \mid e_S\, e_S \mid op^1\, e_S \mid op^2\, e_S\, e_S \mid \text{dyn}\ \tau\ e_D$$
$$\tau = \text{Int} \mid \text{Nat} \mid \tau{\times}\tau \mid \tau \Rightarrow \tau$$
$$i \in \mathbb{Z}$$
$$op^1 = \text{fst} \mid \text{snd}$$
$$op^2 = \text{sum} \mid \text{quotient}$$

Fig. 1. Twin languages syntax

Typing Syntax extends Surface Syntax $\boxed{\tau \leqslant{:}\ \tau}$

$$e = e_S \mid e_D \mid \text{Err}$$
$$\text{Err} = \text{BndryErr} \mid \text{TagErr}$$
$$\Gamma = \cdot \mid x, \Gamma \mid (x{:}\tau), \Gamma$$

$$\overline{\text{Nat} \leqslant{:}\ \text{Int}}$$

$\boxed{\Delta : op^1 {\times} \tau \longrightarrow \tau}$

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

$$\frac{\tau'_d \leqslant{:}\ \tau_d \quad \tau_c \leqslant{:}\ \tau'_c}{\tau_d \Rightarrow \tau_c \leqslant{:}\ \tau'_d \Rightarrow \tau'_c} \qquad \frac{\tau_0 \leqslant{:}\ \tau'_0 \quad \tau_1 \leqslant{:}\ \tau'_1}{\tau_0 {\times} \tau_1 \leqslant{:}\ \tau'_0 {\times} \tau'_1}$$

$\boxed{\Delta : op^2 {\times} \tau {\times} \tau \longrightarrow \tau}$

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

$$\frac{}{\tau \leqslant{:}\ \tau} \qquad \frac{\tau \leqslant{:}\ \tau' \quad \tau' \leqslant{:}\ \tau''}{\tau \leqslant{:}\ \tau''}$$

$\boxed{\Gamma \vdash e}$

$$\frac{x \in \Gamma}{\Gamma \vdash x} \quad \frac{x, \Gamma \vdash e}{\Gamma \vdash \lambda x.\, e} \quad \frac{}{\Gamma \vdash i} \quad \frac{\Gamma \vdash e_0 \quad \Gamma \vdash e_1}{\Gamma \vdash \langle e_0, e_1 \rangle} \quad \frac{\Gamma \vdash e_0 \quad \Gamma \vdash e_1}{\Gamma \vdash e_0\, e_1} \quad \frac{\Gamma \vdash e}{\Gamma \vdash op^1\, e} \quad \frac{\Gamma \vdash e_0 \quad \Gamma \vdash e_1}{\Gamma \vdash op^2\, e_0\, e_1}$$

$$\frac{}{\Gamma \vdash \text{Err}} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{stat}\ \tau\ e}$$

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

$$\frac{(x{:}\tau) \in \Gamma}{\Gamma \vdash x : \tau} \quad \frac{(x{:}\tau_d), \Gamma \vdash e : \tau_c}{\Gamma \vdash \lambda(x{:}\tau_d).\, e : \tau_d \Rightarrow \tau_c} \quad \frac{i \in \mathbb{N}}{\Gamma \vdash i : \text{Nat}} \quad \frac{}{\Gamma \vdash i : \text{Int}} \quad \frac{\Gamma \vdash e_0 : \tau_0 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \langle e_0, e_1 \rangle : \tau_0 {\times} \tau_1}$$

$$\frac{\Gamma \vdash e_0 : \tau_d \Rightarrow \tau_c \quad \Gamma \vdash e_1 : \tau_d}{\Gamma \vdash e_0\, e_1 : \tau_c} \quad \frac{\Gamma \vdash e_0 : \tau_0 \quad \Delta(op^1, \tau_0) = \tau}{\Gamma \vdash op^1\, e_0 : \tau} \quad \frac{\Gamma \vdash e_0 : \tau_0 \quad \Gamma \vdash e_1 : \tau_1 \quad \Delta(op^2, \tau_0, \tau_1) = \tau}{\Gamma \vdash op^2\, e_0\, e_1 : \tau} \quad \frac{\Gamma \vdash e : \tau' \quad \tau' \leqslant{:}\ \tau}{\Gamma \vdash e : \tau}$$

$$\frac{}{\Gamma \vdash \text{Err} : \tau} \qquad \frac{\Gamma \vdash e}{\Gamma \vdash \text{dyn}\ \tau\ e : \tau}$$

Fig. 2. Twin languages static typing judgments

Figure 2 presents a relatively straightforward typing system for the complete syntax, augmented with error terms. To accomodate the two kinds of expressions, there are two typing judgments. The first judgment, $\Gamma \vdash e_D$, essentially states that the expression e is closed; this weak property exemplifies the ahead-of-time checking available in some dynamically-typed languages. The second judgment, $\Gamma \vdash e_S : \tau$, is mostly conventional. Following the grammars, both judgments come with rules for boundary terms, which refer to the opposite judgment on their subexpressions. For example, $\Gamma \vdash$ stat τ e holds only if the enclosed expression is well-typed.

The Δ function assigns a type to the primitives. The subtyping judgment ($\leqslant:$) is based on the subset relation between natural numbers and integers. Subtyping adds a logical distinction to the type system that is not automatically enforced by the host language or the primitives.

2.2 Common Semantic Notions

Figure 3 introduces the common semantic notions. The syntactic components of this figure are expressions e, values v, irreducible results r, and two kinds of evaluation context. A boundary-free context $E^{\bullet}$ does not contain dyn or stat boundary terms but a multi-language context E may.

The semantic components in figure 3 are the δ function and the $\triangleright_S$ and $\triangleright_D$ notions of reduction [8]. The δ function is a partial mathematical specification for the primitives. The partial nature of δ represents certain errors that the use of a primitive operation may trigger; we assume δ is computable wherever it is defined for an input. Specifically, primitive operations give rise to two kinds of errors:

- The semantic models reduce a program to a *tag error* when a primitive operation is applied to inappropriate values. Mathematically speaking, the δ function is undefined for the values. The name alludes to the idea that (virtual or abstract) machines represent one form of value differently from others, e.g., pointers to functions have different type-tag bits than integers. Thus, the machine is able to report the addition of a function to an integer as a tag mismatch.
- By contrast, a *boundary error* is the result of applying a partial primitive operation, such as division, to exceptional inputs. Division-by-zero is a representative example. The δ function is defined for the inputs, and its result represents a boundary error. The name "boundary error" suggests that one part of the program received an incorrect value from another part; in the case of δ, the run-time library (which implements the primitives directly as hardware instructions) received the value from (possibly-typed) user code. Naturally, the same kind of error may arise when typed and untyped regions of code interact.

The functions Δ and δ satisfy the typability condition [88].

Proposition 2.0 : δ *typability*

- If $\vdash v : \tau_v$ and $\Delta(op^1, \tau_v) = \tau$ then $\vdash \delta(op^1, v) : \tau$.
- If $\vdash v_0 : \tau_0$ and $\vdash v_1 : \tau_1$ and $\Delta(op^2, \tau_0, \tau_1) = \tau$ then $\vdash \delta(op^2, v_0, v_1) : \tau$.

The notion of reduction $\triangleright_S$ defines a semantics for statically-typed expressions. It relates the left-hand side to the right-hand side on an unconditional basis, which expresses the reliance on the type system to prevent stuck terms up front. The notion of reduction $\triangleright_D$ defines a semantics for dynamically-typed expressions. A dynamic expression may attempt to apply an integer or send inappropriate arguments to a primitive operation. Hence, $\triangleright_D$ explicitly checks for malformed expressions and signals a tag error. These checks make the untyped language safe.

The three models in the following three sections build upon figure 3. They define a pair of *boundary functions* ($\mathcal{D}$ and $\mathcal{S}$) for transporting a value across a boundary term, extend the $\triangleright_S$ and $\triangleright_D$ notions of reduction, and syntactically close the notions of reduction to reduction relations $\rightarrow_S^*$ and $\rightarrow_D^*$ for multi-language evaluation contexts. That is, $\rightarrow_S^*$ and $\rightarrow_D^*$ reduce terms whose root is produced by e_S and e_D, respectively.

| Evaluation Syntax | extends Surface Syntax

$e \quad = x \mid v \mid \langle e, e \rangle \mid e\, e \mid op^1\, e \mid op^2\, e\, e \mid \mathsf{dyn}\, \tau\, e \mid \mathsf{stat}\, \tau\, e \mid \mathsf{Err}$

$v \quad = i \mid \langle v, v \rangle \mid \lambda x.\, e \mid \lambda(x{:}\tau).\, e$

$\mathsf{Err} = \mathsf{BndryErr} \mid \mathsf{TagErr}$

$r \quad = v \mid \mathsf{Err}$

$E^\bullet = [\,] \mid E^\bullet\, e \mid v\, E^\bullet \mid \langle E^\bullet, e \rangle \mid \langle v, E^\bullet \rangle \mid op^1\, E^\bullet \mid op^2\, E^\bullet\, e \mid op^2\, v\, E^\bullet$

$E \quad = E^\bullet \mid E\, e \ \mid v\, E \ \mid \langle E, e \rangle \mid \langle v, E \rangle \mid op^1\, E \mid op^2\, E\, e \mid op^2\, v\, E \mid \mathsf{dyn}\, \tau\, E \mid \mathsf{stat}\, \tau\, E$

| $\delta : op^1 {\times} v \longrightarrow v$

$\delta(\mathsf{fst}, \langle v_0, v_1 \rangle) \ = v_0$

$\delta(\mathsf{snd}, \langle v_0, v_1 \rangle) = v_1$

| $\delta : op^2 {\times} v {\times} v \longrightarrow r$

$\delta(\mathsf{sum}, i_0, i_1) \qquad = i_0 + i_1$

$\delta(\mathsf{quotient}, i_0, 0) = \mathsf{BndryErr}$

$\delta(\mathsf{quotient}, i_0, i_1) = \lfloor i_0/i_1 \rfloor$
$\qquad$ if $i_1 \neq 0$

| $e \rhd_{\mathsf{S}} e$

$(\lambda(x{:}\tau).\, e)\, v \ \rhd_{\mathsf{S}}\ e[x \leftarrow v]$

$op^1\, v \qquad\qquad \rhd_{\mathsf{S}}\ \delta(op^1, v)$

$op^2\, v_0\, v_1 \qquad \rhd_{\mathsf{S}}\ \delta(op^2, v_0, v_1)$

| $e \rhd_{\mathsf{D}} e$

$v_0\, v_1 \qquad\quad \rhd_{\mathsf{D}}\ \mathsf{TagErr}$
$\quad$ if $v_0 \in \mathbb{Z}$ or $v_0 = \langle v, v' \rangle$

$(\lambda x.\, e)\, v \ \ \rhd_{\mathsf{D}}\ e[x \leftarrow v]$

$op^1\, v \qquad\ \rhd_{\mathsf{D}}\ \mathsf{TagErr}$
$\quad$ if $\delta(op^1, v)$ is undefined

$op^1\, v \qquad\ \rhd_{\mathsf{D}}\ \delta(op^1, v)$

$op^2\, v_0\, v_1 \ \rhd_{\mathsf{D}}\ \mathsf{TagErr}$
$\quad$ if $\delta(op^2, v_0, v_1)$ is undefined

$op^2\, v_0\, v_1 \ \rhd_{\mathsf{D}}\ \delta(op^2, v_0, v_1)$

Fig. 3. Common semantic notions

2.3 Higher-Order Embedding

The higher-order embedding is based on the idea that types enforce levels of abstraction [60]. In a typed language, the type checker ensures that the whole program respects these abstractions. A migratory typing system can provide a similar guarantee if the semantics dynamically enforces a type specification on every untyped value that enters a typed context.

The higher-order embedding uses a type-directed strategy to transport a value across a type boundary. If an untyped value meets a boundary that expects a value of a base type, such as Int, then the strategy is to check the shape of the value. If the boundary expects a value of an algebraic type, such as a pair, then the strategy is to check the value and recursively transport its components. Lastly, if the boundary expects a value of a higher type, such as $(\mathsf{Nat} \Rightarrow \mathsf{Nat})$, then the strategy is to check the constructor and monitor the future interactions between the value and the context. For the specific case of an untyped function f and the type $(\mathsf{Nat} \Rightarrow \mathsf{Nat})$, the higher-order embedding wraps f in a proxy. The wrapper checks that every result computed by f is of type Nat and otherwise halts the program with a witness that f does not match the type.

2.3.1 Model. Figure 4 presents a model of the higher-order embedding. Its centerpiece is the pair of boundary functions: $\mathcal{D}_{\mathsf{H}}$ and $\mathcal{S}_{\mathsf{H}}$. The $\mathcal{D}_{\mathsf{H}}$ function imports a dynamically-typed value into a statically-typed context by checking the shape of the value and proceeding as outlined above. In particular, $\mathcal{D}_{\mathsf{H}}$ transports an untyped value v into a context expecting a function with

$\boxed{\text{Language H}}$ extends Evaluation Syntax

$v \ = \ \dots \mid \text{mon} \, (\tau \Rightarrow \tau) \, v$

$\boxed{\mathcal{D}_{\text{H}} : \tau \times v \longrightarrow e}$

$\mathcal{D}_{\text{H}}(\tau_d \Rightarrow \tau_c, v) \qquad = \text{mon} \, (\tau_d \Rightarrow \tau_c) \, v$
  if $v = \lambda x. \, e$ or $v = \text{mon} \, \tau' \, v'$
$\mathcal{D}_{\text{H}}(\tau_0 \times \tau_1, \langle v_0, v_1 \rangle) = \langle \text{dyn} \, \tau_0 \, v_0, \text{dyn} \, \tau_1 \, v_1 \rangle$
$\mathcal{D}_{\text{H}}(\text{Int}, i) \qquad\quad = i$
$\mathcal{D}_{\text{H}}(\text{Nat}, i) \qquad\quad = i$
  if $i \in \mathbb{N}$
$\mathcal{D}_{\text{H}}(\tau, v) \qquad\qquad = \text{BndryErr}$
  otherwise

$\boxed{\mathcal{S}_{\text{H}} : \tau \times v \longrightarrow e}$

$\mathcal{S}_{\text{H}}(\tau_d \Rightarrow \tau_c, v) \qquad = \text{mon} \, (\tau_d \Rightarrow \tau_c) \, v$

$\mathcal{S}_{\text{H}}(\tau_0 \times \tau_1, \langle v_0, v_1 \rangle) = \langle \text{stat} \, \tau_0 \, v_0, \text{stat} \, \tau_1 \, v_1 \rangle$
$\mathcal{S}_{\text{H}}(\text{Int}, v) \qquad\quad = v$
$\mathcal{S}_{\text{H}}(\text{Nat}, v) \qquad\quad = v$

$\boxed{e \vartriangleright_{\text{H-S}} e}$ extends $\vartriangleright_{\text{S}}$

$\text{dyn} \, \tau \, v \qquad\qquad\qquad \vartriangleright_{\text{H-S}} \ \mathcal{D}_{\text{H}}(\tau, v)$
$(\text{mon} \, (\tau_d \Rightarrow \tau_c) \, v_f) \, v \ \vartriangleright_{\text{H-S}} \ \text{dyn} \, \tau_c \, (v_f \, e')$
  where $e' = \text{stat} \, \tau_d \, v$

$\boxed{e \vartriangleright_{\text{H-D}} e}$ extends $\vartriangleright_{\text{D}}$

$\text{stat} \, \tau \, v \qquad\qquad\qquad \vartriangleright_{\text{H-D}} \ \mathcal{S}_{\text{H}}(\tau, v)$
$(\text{mon} \, (\tau_d \Rightarrow \tau_c) \, v_f) \, v \ \vartriangleright_{\text{H-D}} \ \text{stat} \, \tau_c \, (v_f \, e')$
  where $e' = \text{dyn} \, \tau_d \, v$

$\boxed{e \rightarrow^{*}_{\text{H-S}} e}$ reflexive, transitive closure of $\rightarrow_{\text{H-S}}$

$E^{\bullet}[e] \qquad\qquad \rightarrow_{\text{H-S}} \ E^{\bullet}[e']$
  if $e \vartriangleright_{\text{H-S}} e'$
$E[\text{stat} \, \tau \, E^{\bullet}[e]] \ \rightarrow_{\text{H-S}} \ E[\text{stat} \, \tau \, E^{\bullet}[e']]$
  if $e \vartriangleright_{\text{H-S}} e'$
$E[\text{dyn} \, \tau \, E^{\bullet}[e]] \ \rightarrow_{\text{H-S}} \ E[\text{dyn} \, \tau \, E^{\bullet}[e']]$
  if $e \vartriangleright_{\text{H-D}} e'$
$E[\text{Err}] \qquad\qquad \rightarrow_{\text{H-S}} \ \text{Err}$

$\boxed{e \rightarrow^{*}_{\text{H-D}} e}$ reflexive, transitive closure of $\rightarrow_{\text{H-D}}$

$E^{\bullet}[e] \qquad\qquad \rightarrow_{\text{H-D}} \ E^{\bullet}[e']$
  if $e \vartriangleright_{\text{H-D}} e'$
$E[\text{stat} \, \tau \, E^{\bullet}[e]] \ \rightarrow_{\text{H-D}} \ E[\text{stat} \, \tau \, E^{\bullet}[e']]$
  if $e \vartriangleright_{\text{H-S}} e'$
$E[\text{dyn} \, \tau \, E^{\bullet}[e]] \ \rightarrow_{\text{H-D}} \ E[\text{dyn} \, \tau \, E^{\bullet}[e']]$
  if $e \vartriangleright_{\text{H-D}} e'$
$E[\text{Err}] \qquad\qquad \rightarrow_{\text{H-D}} \ \text{Err}$

Fig. 4. Higher-Order Embedding

domain τ_d and codomain τ_c by checking that v is a function and creating a monitor of the shape $(\text{mon} \, (\tau_d \Rightarrow \tau_c) \, v)$. Conversely, the $\mathcal{S}_{\text{H}}$ function exports a typed value to an untyped context. It transports an integer as-is, transports a pair recursively, and wraps a typed function in a monitor to protect it against untyped arguments.

The extended notions of reduction in figure 4 define the semantics of boundary-crossing and monitors. In a statically-typed context, the application of a monitor expresses the application of a dynamically-typed function to a typed argument. Thus the semantics unfolds the monitor into two boundary terms: a dyn boundary in which to apply the dynamically-typed function and an inner stat expression to transport the argument. In a dynamically-typed context, a monitor encapsulates a typed function and application unfolds into two dual boundary terms.

The boundary functions and the notions of reductions together define the semantics of mixed-typed expressions. There are two main reduction relations: $\rightarrow^{*}_{\text{H-S}}$ for typed expressions and $\rightarrow^{*}_{\text{H-D}}$ for untyped expressions. The only difference between the two is how they act on an expression that does not contain boundary terms. The typed reduction relation steps via $\vartriangleright_{\text{S}}$ by default, and the untyped relation steps via $\vartriangleright_{\text{D}}$ by default. For other cases, the relations are identical.

$$\boxed{\Gamma \vdash_{\mathsf{H}} e} \text{ extends } \Gamma \vdash e \qquad\qquad\qquad \boxed{\Gamma \vdash_{\mathsf{H}} e : \tau} \text{ extends } \Gamma \vdash e : \tau$$

$$\frac{\Gamma \vdash_{\mathsf{H}} v : \tau_d \Rightarrow \tau_c}{\Gamma \vdash_{\mathsf{H}} \mathsf{mon}\,(\tau_d \Rightarrow \tau_c)\,v} \qquad\qquad \frac{\Gamma \vdash_{\mathsf{H}} v}{\Gamma \vdash_{\mathsf{H}} \mathsf{mon}\,(\tau_d \Rightarrow \tau_c)\,v : \tau_d \Rightarrow \tau_c}$$

Fig. 5. Property judgments for the higher-order embedding

2.3.2 Soundness. Figure 5 presents two properties for the higher-order embedding: one for dynamically-typed expressions and one for statically-typed expressions. Each property extends the corresponding judgment from figure 2 with a rule for monitors. The property for dynamic expressions (in the left column) states that a typed value may be wrapped in a monitor of the same type. The static property states that any untyped value may be wrapped in a monitor.

The soundness theorems for the higher-order embedding state three results about surface-language expressions: (1) reduction is fully defined, (2) reduction in a statically-typed context cannot raise a tag error, and (3) reduction preserves the properties from figure 5.

Theorem 2.1 : *static H-soundness*
If $e \in e_S$ and $\vdash e : \tau$
then $\vdash_{\mathsf{H}} e : \tau$ and one of the following holds:
- $e \rightarrow^*_{\mathsf{H\text{-}S}} v$ and $\vdash_{\mathsf{H}} v : \tau$
- $e \rightarrow^*_{\mathsf{H\text{-}S}} E[\mathsf{dyn}\,\tau'\,E^\bullet[e']]$ and $e' \rhd_{\mathsf{H\text{-}D}} \mathsf{TagErr}$
- $e \rightarrow^*_{\mathsf{H\text{-}S}} \mathsf{BndryErr}$
- e diverges

Theorem 2.2 : *dynamic H-soundness*
If $e \in e_D$ and $\vdash e$
then $\vdash_{\mathsf{H}} e$ and one of the following holds:
- $e \rightarrow^*_{\mathsf{H\text{-}D}} v$ and $\vdash_{\mathsf{H}} v$
- $e \rightarrow^*_{\mathsf{H\text{-}D}} E[e']$ and $e' \rhd_{\mathsf{H\text{-}D}} \mathsf{TagErr}$
- $e \rightarrow^*_{\mathsf{H\text{-}D}} \mathsf{BndryErr}$
- e diverges

Proof (sketch): First, $\vdash e : \tau$ implies $\vdash_{\mathsf{H}} e : \tau$ (similarly for the dynamic property) because the latter judgment generalizes the former. The rest follows from progress and preservation lemmas [31]. □

One notable lemma for the proof states that the codomain of the $\mathcal{D}_{\mathsf{H}}$ boundary function is typed.

Lemma 2.3 : $\mathcal{D}_{\mathsf{H}}$ *soundness*
If $\Gamma \vdash_{\mathsf{H}} v$ and $\mathcal{D}_{\mathsf{H}}(\tau, v) = e$ then $\Gamma \vdash_{\mathsf{H}} e : \tau$

A similar lemma does not hold of the surface-language typing judgment. Let v be the identity function $(\lambda x.\, x)$. In this case $\vdash v$ holds but $\mathcal{D}_{\mathsf{H}}((\mathsf{Int} \Rightarrow \mathsf{Int}), v)$ returns a monitor, which is not part of the surface language. A language with mutable references would require a similar extension to monitor reads and writes [72].

2.4 Erasure Embedding

The erasure approach is based on a view of types as an optional syntactic artifact. From this perspective, type annotations are just a structured form of comment that help developers read a codebase. A secondary purpose is to enable IDE tools. Whether the types are sound is incidental.

The justification for the erasure point of view is that the host language safely executes untyped code. If all code is treated as untyped, there is no risk of undefined behavior.

2.4.1 Model. Figure 6 presents a semantics for erasure. The two boundary functions, $\mathcal{D}_{\mathsf{E}}$ and $\mathcal{S}_{\mathsf{E}}$, let values freely cross type boundaries. The two notions of reduction must therefore accomodate values from the opposite grammar. The static notion of reduction $\rhd_{\mathsf{E\text{-}S}}$ allows the application of dynamically-typed functions and, in contrast to $\rhd_{\mathsf{H\text{-}S}}$, must check the validity of arguments to primitives. The dynamic notion of reduction $\rhd_{\mathsf{E\text{-}D}}$ allows the application of statically-typed functions. The reduction relations $\rightarrow^*_{\mathsf{E\text{-}S}}$ and $\rightarrow^*_{\mathsf{E\text{-}D}}$ are based on the compatible closure of the matching notion of reduction.

$\boxed{\mathcal{D}_\mathsf{E} : \tau \times v \longrightarrow e}$

$\mathcal{D}_\mathsf{E}(\tau, v) = v$

$\boxed{\mathcal{S}_\mathsf{E} : \tau \times v \longrightarrow e}$

$\mathcal{S}_\mathsf{E}(\tau, v) = v$

$\boxed{e \rhd_{\mathsf{E\text{-}S}} e}$ extends $\rhd_\mathsf{S}$

$\begin{aligned}
\mathsf{dyn}\ \tau\ v &\quad \rhd_{\mathsf{E\text{-}S}}\ \mathcal{D}_\mathsf{E}(\tau, v) \\
\mathsf{stat}\ \tau\ v &\quad \rhd_{\mathsf{E\text{-}S}}\ \mathcal{S}_\mathsf{E}(\tau, v) \\
(\lambda x.\, e)\ v &\quad \rhd_{\mathsf{E\text{-}S}}\ e[x \leftarrow v] \\
v_0\ v_1 &\quad \rhd_{\mathsf{E\text{-}S}}\ \mathsf{TagErr} \\
&\quad \text{if } v_0 \in \mathbb{Z} \text{ or } v_0 = \langle v, v' \rangle \\
op^1\ v &\quad \rhd_{\mathsf{E\text{-}S}}\ \mathsf{TagErr} \\
&\quad \text{if } \delta(op^1, v) \text{ is undefined} \\
op^2\ v_0\ v_1 &\quad \rhd_{\mathsf{E\text{-}S}}\ \mathsf{TagErr} \\
&\quad \text{if } \delta(op^2, v_0, v_1) \text{ is undefined}
\end{aligned}$

$\boxed{e \rhd_{\mathsf{E\text{-}D}} e}$ extends $\rhd_\mathsf{D}$

$\begin{aligned}
\mathsf{stat}\ \tau\ v &\quad \rhd_{\mathsf{E\text{-}D}}\ \mathcal{S}_\mathsf{E}(\tau, v) \\
\mathsf{dyn}\ \tau\ v &\quad \rhd_{\mathsf{E\text{-}D}}\ \mathcal{D}_\mathsf{E}(\tau, v) \\
(\lambda(x{:}\tau_d).\, e)\ v &\quad \rhd_{\mathsf{E\text{-}D}}\ e[x \leftarrow v]
\end{aligned}$

$\boxed{e \rightarrow^*_{\mathsf{E\text{-}S}} e}$ reflexive, transitive closure of $\rightarrow_{\mathsf{E\text{-}S}}$

$\begin{aligned}
E[e] &\quad \rightarrow_{\mathsf{E\text{-}S}}\ E[e'] \\
&\quad \text{if } e \rhd_{\mathsf{E\text{-}S}} e' \\
E[\mathsf{Err}] &\quad \rightarrow_{\mathsf{E\text{-}S}}\ \mathsf{Err}
\end{aligned}$

$\boxed{e \rightarrow^*_{\mathsf{E\text{-}D}} e}$ reflexive, transitive closure of $\rightarrow_{\mathsf{E\text{-}D}}$

$\begin{aligned}
E[e] &\quad \rightarrow_{\mathsf{E\text{-}D}}\ E[e'] \\
&\quad \text{if } e \rhd_{\mathsf{E\text{-}D}} e' \\
E[\mathsf{Err}] &\quad \rightarrow_{\mathsf{E\text{-}D}}\ \mathsf{Err}
\end{aligned}$

Fig. 6. Erasure Embedding

$\boxed{\Gamma \vdash_\mathsf{E} e}$ (selected rules)

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

Fig. 7. Common property judgment for the erasure embedding

2.4.2 Soundness. Figure 7 extends the judgment for a well-formed dynamically-typed expression to accomodate type-annotated expressions. This judgment ignores the type annotations; for any expression e, the judgment $\vdash_\mathsf{E} e$ holds if e is closed. Soundness for the erasure embedding states that reduction is well-defined for statically-typed and dynamically-typed expressions.

Theorem 2.4 : *static* E*-soundness*

If $e \in e_\mathsf{S}$ and $\vdash e : \tau$
then $\vdash_\mathsf{E} e$ and one of the following holds:
- $e \rightarrow^*_{\mathsf{E\text{-}S}} v$ and $\vdash_\mathsf{E} v$
- $e \rightarrow^*_{\mathsf{E\text{-}S}} \mathsf{TagErr}$
- $e \rightarrow^*_{\mathsf{E\text{-}S}} \mathsf{BndryErr}$
- e diverges

Theorem 2.5 : *dynamic* E*-soundness*

If $e \in e_\mathsf{D}$ and $\vdash e$
then $\vdash_\mathsf{E} e$ and one of the following holds:
- $e \rightarrow^*_{\mathsf{E\text{-}D}} v$ and $\vdash_\mathsf{E} v$
- $e \rightarrow^*_{\mathsf{E\text{-}D}} \mathsf{TagErr}$
- $e \rightarrow^*_{\mathsf{E\text{-}D}} \mathsf{BndryErr}$
- e diverges

Proof (sketch): A well-typed term is closed, therefore $\vdash e : \tau$ implies that $\vdash_\mathsf{E} e$ holds. The rest follows from progress and preservation lemmas [31]. $\square$

The erasure embedding clearly ignores the types in a mixed-typed expression. A simple example is the expression $(\mathsf{dyn}\ \mathsf{Int}\ \langle 2, 2 \rangle)$, which has the static type Int but reduces to a pair. The embedding is sound, however, for well-typed expressions that do not contain boundary terms. In other words,

a disciplined programmer who avoids libraries without types may be justified in assuming that evaluation preserves static types and never results in a tag error:

Theorem 2.6 : *boundary-free* E-*soundness*

If $e \in e_S$ and $\vdash e : \tau$ and e does not contain a subexpression (dyn τ' e')

then one of the following holds:

- $e \rightarrow^*_{\text{E-S}} v$ and $\vdash v : \tau$
- $e \rightarrow^*_{\text{E-S}}$ BndryErr
- e diverges

Proof (sketch): By progress and preservation lemmas [31]. □

2.5 First-Order Embedding

The first-order approach is the result of two assumptions: one philosophical, one pragmatic. The philosophical assumption is that the purpose of types is to prevent evaluation from "going wrong" [49] in the sense of applying a typed elimination form to a value outside its domain. In particular, the elimination forms in our surface language are function application and primitive application. A function application (v_0 v_1) expects v_0 to be a function; primitive application expects arguments for which δ is defined. The goal of the first-order embedding is to ensure that such basic assumptions are always satisfied in typed contexts so that typed execution cannot get stuck.

The pragmatic assumption is that run-time monitoring is impractical. For one, implementing monitors requires a significant engineering effort [72]. Such monitors must preserve all the observations that dynamically-typed code can make of the original value, including object-identity tests. Second, monitoring adds a significant run-time cost [33, 76].

Based on these assumptions, the first-order semantics employs a type-directed rewriting pass over typed code to defend against untyped inputs. The defense takes the form of type-constructor checks; for example, if a typed context expects a value of type (Nat $\Rightarrow$ Nat) then a run-time check ensures that the context receives a function. If this function is applied in a context expecting a Nat, then a second run-time check confirms that the result is a natural number. If the same function is applied *in a different typed context* that expects a result of type (Int×Int), then a different run-time check confirms that the result is a pair.

Constructor checks run without creating monitors, work in near-constant time,[1] and ensure that every value in a typed context has the correct top-level shape. Since the notions of reduction rely on *only* the shape of a value to avoid stuck states, well-typed programs cannot "go wrong."

2.5.1 Model. Figure 8 presents a model of the first-order approach. The model represents a type-constructor check as a chk expression; informally, the semantics of (chk K e) is to reduce e to a value and affirm that it matches the K constructor. Type constructors K include one constructor $\lfloor \tau \rfloor$ for each type τ, and the technical Any constructor, which does not correspond to a static type.

The specific purpose of Any is to reflect the weak invariants of the first-order semantics. In contrast to full types, type constructors say nothing about the contents of a structured value. The first and second components of a generic Pair value can have any shape, and similarly the result of applying a function of constructor Fun can be any value.[2] Put another way, the Any constructor is necessary because information about type constructors is not compositional.

In the model, the above-mentioned rewriting pass corresponds to the judgment $\Gamma \vdash e : \tau \rightsquigarrow e'$, which states that e' is the *completion* [36] of the surface language expression. The rewritten

---

[1]The constructor check for a union type or structural object type require time linear in the size of the type.

[2]Since the contractum of function application is an expression, the model includes the "no-op" boundary term (dyn e) to support the application of an untyped function in a typed context. The (stat e) boundary serves a dual purpose. These two forms facilitate the proofs of the progress and preservation lemmas. They need not appear in an implementation.

$\boxed{\text{Language 1}}$ extends Evaluation Syntax	$\boxed{\lfloor \cdot \rfloor : \tau \longrightarrow K}$		
$e \quad = \ldots \mid \text{chk } K\ e \mid \text{dyn } e \mid \text{stat } e$	$\lfloor \text{Int} \rfloor$	$=$	Int
$E^{\bullet} = \ldots \mid \text{chk } K\ E^{\bullet}$	$\lfloor \text{Nat} \rfloor$	$=$	Nat
$E \quad = \ldots \mid \text{chk } K\ E \mid \text{dyn } E \mid \text{stat } E$	$\lfloor \tau_0 {\times} \tau_1 \rfloor$	$=$	Pair
$K \quad = \text{Int} \mid \text{Nat} \mid \text{Pair} \mid \text{Fun} \mid \text{Any}$	$\lfloor \tau_d {\Rightarrow} \tau_c \rfloor$	$=$	Fun

$\boxed{\Gamma \vdash e : \tau \rightsquigarrow e}$ (selected rules)

$$\frac{\Gamma \vdash e_0 : \tau_d {\Rightarrow} \tau_c \rightsquigarrow e_0' \quad \Gamma \vdash e_1 : \tau_d \rightsquigarrow e_1'}{\Gamma \vdash e_0\ e_1 : \tau_c \rightsquigarrow \text{chk } \lfloor \tau_c \rfloor\ (e_0'\ e_1')} \qquad \frac{\Gamma \vdash e : \tau_0 {\times} \tau_1 \rightsquigarrow e'}{\Gamma \vdash \text{fst } e : \tau_0 \rightsquigarrow \text{chk } \lfloor \tau_0 \rfloor\ (\text{fst } e')}$$

$\boxed{\mathcal{D}_1 : \tau {\times} v \longrightarrow v}$ $\boxed{\mathcal{S}_1 : \tau {\times} v \longrightarrow v}$
$\mathcal{D}_1(\tau, v) = \mathcal{X}(\lfloor \tau \rfloor, v)$ $\mathcal{S}_1(\tau, v) = v$

$\boxed{\mathcal{X} : K {\times} v \longrightarrow v}$

$\mathcal{X}(\text{Fun}, \lambda x.\, e)$	$= \lambda x.\, e$		$\mathcal{X}(\text{Int}, i)$	$= i$
$\mathcal{X}(\text{Fun}, \lambda(x{:}\tau).\, e)$	$= \lambda(x{:}\tau).\, e$		$\mathcal{X}(\text{Nat}, i)$	$= i$
$\mathcal{X}(\text{Pair}, \langle v_0, v_1 \rangle)$	$= \langle v_0, v_1 \rangle$			$\text{if } i \in \mathbb{N}$
			$\mathcal{X}(K, v)$	$= \text{BndryErr}$
				otherwise

$\boxed{e \rhd_{\text{1-S}} e}$ extends and overrides $\rhd_{\text{S}}$ $\boxed{e \rhd_{\text{1-D}} e}$ extends $\rhd_{\text{D}}$

$\text{dyn } v$	$\rhd_{\text{1-S}}$	v		$\text{stat } v$	$\rhd_{\text{1-D}}$	v
$\text{dyn } \tau\ v$	$\rhd_{\text{1-S}}$	$\mathcal{D}(\tau, v)$		$\text{stat } \tau\ v$	$\rhd_{\text{1-D}}$	$\mathcal{S}(\tau, v)$
$\text{chk } K\ v$	$\rhd_{\text{1-S}}$	$\mathcal{X}(K, v)$				

$(\lambda(x{:}\tau).\, e)\ v \rhd_{\text{1-S}} \text{BndryErr}$ $(\lambda(x{:}\tau).\, e)\ v \rhd_{\text{1-D}} \text{BndryErr}$
  $\text{if } \mathcal{X}(\lfloor \tau \rfloor, v) = \text{BndryErr}$ $\text{if } \mathcal{X}(\lfloor \tau \rfloor, v) = \text{BndryErr}$
$(\lambda(x{:}\tau).\, e)\ v \rhd_{\text{1-S}} e[x \leftarrow \mathcal{X}(\lfloor \tau \rfloor, v)]$ $(\lambda(x{:}\tau).\, e)\ v \rhd_{\text{1-D}} \text{stat } (e[x \leftarrow \mathcal{X}(\lfloor \tau \rfloor, v)])$
  $\text{if } \mathcal{X}(\lfloor \tau \rfloor, v) \neq \text{BndryErr}$ $\text{if } \mathcal{X}(\lfloor \tau \rfloor, v) \neq \text{BndryErr}$
$(\lambda x.\, e)\ v \quad \rhd_{\text{1-S}} \text{dyn } (e[x \leftarrow v])$

$\boxed{e \rightarrow^{*}_{\text{1-S}} e}$ reflexive, transitive closure of $\rightarrow_{\text{1-S}}$ $\boxed{e \rightarrow^{*}_{\text{1-D}} e}$ reflexive, transitive closure of $\rightarrow_{\text{1-D}}$

$E^{\bullet}[e]$	$\rightarrow_{\text{1-S}} E^{\bullet}[e']$		$E^{\bullet}[e]$	$\rightarrow_{\text{1-D}} E^{\bullet}[e']$
$\quad \text{if } e \rhd_{\text{1-S}} e'$			$\quad \text{if } e \rhd_{\text{1-D}} e'$	
$E[\text{stat } \tau\ E^{\bullet}[e]]$	$\rightarrow_{\text{1-S}} E[\text{stat } \tau\ E^{\bullet}[e']]$		$E[\text{stat } \tau\ E^{\bullet}[e]]$	$\rightarrow_{\text{1-D}} E[\text{stat } \tau\ E^{\bullet}[e']]$
$\quad \text{if } e \rhd_{\text{1-S}} e'$			$\quad \text{if } e \rhd_{\text{1-S}} e'$	
$E[\text{dyn } \tau\ E^{\bullet}[e]]$	$\rightarrow_{\text{1-S}} E[\text{dyn } \tau\ E^{\bullet}[e']]$		$E[\text{dyn } \tau\ E^{\bullet}[e]]$	$\rightarrow_{\text{1-D}} E[\text{dyn } \tau\ E^{\bullet}[e']]$
$\quad \text{if } e \rhd_{\text{1-D}} e'$			$\quad \text{if } e \rhd_{\text{1-D}} e'$	
$E[\text{Err}]$	$\rightarrow_{\text{1-S}} \text{Err}$		$E[\text{Err}]$	$\rightarrow_{\text{1-D}} \text{Err}$

Fig. 8. First-Order Embedding

expression e' includes chk forms around: function applications, fst projections, and snd projections. For any other expression, the result is constructed by structural recursion.

The semantics ensures that every expression of type τ reduces to a value that matches the $\lfloor \tau \rfloor$ constructor. The boundary function $\mathcal{D}_1$ checks that an untyped value entering typed code matches the constructor of the expected type; its implementation defers to the $\mathcal{X}$ boundary-crossing function. The boundary function $\mathcal{S}_1$ lets any typed value—including a function—cross into an untyped context.

The notions of reduction consequently turn the type annotation τ_d on the formal parameter of a typed function $(\lambda(x : \tau_d). e)$ into a check that its actual parameter matches the $\lfloor \tau_d \rfloor$ constructor. In a dynamically-typed context, this check protects a typed function against untyped arguments. In a statically-typed context, this check protects a typed function against mis-matched *typed* arguments; the following example demonstrates the need for this protection by applying a typed function that expects an integer to a typed pair value:

$$\triangle \quad \vdash ((\mathrm{dyn}\,(\mathrm{Int}\times\mathrm{Int}\Rightarrow\mathrm{Int})\,(\mathrm{stat}\,(\mathrm{Int}\Rightarrow\mathrm{Int})\,(\lambda(x:\mathrm{Int}).\,\mathrm{sum}\,x\,x)))\,\langle 0,0\rangle):\mathrm{Int}$$

Note: An alternative way to protect the body of a typed function is to extend the core language with syntax for domain checks [84]. Another alternative is to encode domain checks into the completion of a typed function, in the spirit of $(\lambda(x : \tau_d). e) \rightsquigarrow (\lambda(x : \tau_d). (((\lambda y. \lambda z. z)\,(\mathrm{chk}\,\lfloor \tau_d \rfloor\,x))\,e))$. For the model, we picked the semantic approach to leave the implementation open. **End note.**

2.5.2 Soundness. Figure 9 presents two judgments that express the invariants of the first-order semantics. The first judgment, $\Gamma \vdash_1 e$, applies to untyped expressions. The second judgment is a constructor-typing system that formalizes the intuitions stated above; in particular, the value of a typed variable is guaranteed to match its type constructor, the fst projection can produce any kind of value, and the result of a chk expression matches the given constructor.

Soundness for the first-order embedding states that the evaluation of the *completion* of any surface-level expression preserves the *constructor* of its static type. The theorems furthermore state that only the $\triangleright_{1\text{-D}}$ notion of reduction can yield a tag error, therefore such errors can only occur in dynamically-typed contexts.

Theorem 2.7 : *static 1-soundness*
If $e \in e_S$ and $\vdash e : \tau$
then $\vdash e : \tau \rightsquigarrow e''$ and $\vdash_1 e'' : \lfloor \tau \rfloor$
and one of the following holds:
- $e'' \rightarrow^*_{1\text{-S}} v$ and $\vdash_1 v : \lfloor \tau \rfloor$
- $e'' \rightarrow^*_{1\text{-S}} E[\mathrm{dyn}\,\tau'\,E^\bullet[e']]$ and $e' \triangleright_{1\text{-D}} \mathrm{TagErr}$
- $e'' \rightarrow^*_{1\text{-S}} E[\mathrm{dyn}\,E^\bullet[e']]$ and $e' \triangleright_{1\text{-D}} \mathrm{TagErr}$
- $e'' \rightarrow^*_{1\text{-S}} \mathrm{BndryErr}$
- e'' diverges

Theorem 2.8 : *dynamic 1-soundness*
If $e \in e_D$ and $\vdash e$
then $\vdash e \rightsquigarrow e''$ and $\vdash_1 e''$
and one of the following holds:
- $e'' \rightarrow^*_{1\text{-D}} v$ and $\vdash_1 v$
- $e'' \rightarrow^*_{1\text{-D}} E[e']$ and $e' \triangleright_{1\text{-D}} \mathrm{TagErr}$
- $e'' \rightarrow^*_{1\text{-D}} \mathrm{BndryErr}$
- e'' diverges

Proof (sketch): By progress and preservation lemmas [31]. □

2.6 From Models to Implementations

While the models use two reductions, one for the typed and one for the untyped fragments of code, any practical migratory typing system compiles typed expressions to the host language. In terms of the models, this means $\triangleright_D$ is the only notion of reduction, and statically-typed expressions are rewritten so that $\rightarrow^*_D$ applies. For details, see the supplement [31].

A secondary semantic issue concerns the rules for the application of a typed function in the first-order embedding. As written, the $\triangleright_{1\text{-D}}$ notion of reduction implies a non-standard protocol for function application $(v_0\,v_1)$, namely: (1) check that v_0 is a function; (2) check whether v_0 was defined in typed code; (3) if so, then check v_1 against the static type of v_0. If the host language does not support this protocol, a conservative work-around is to extend the completion judgment to add

$\boxed{K \leqslant: K}$

$$\frac{}{K \leqslant: \mathsf{Any}} \qquad \frac{}{\mathsf{Nat} \leqslant: \mathsf{Int}} \qquad \frac{}{K \leqslant: K} \qquad \frac{K \leqslant: K' \quad K' \leqslant: K''}{K \leqslant: K''}$$

$\boxed{\Gamma \vdash_1 e}$ (selected rules)

$$\frac{x \in \Gamma}{\Gamma \vdash_1 x} \qquad \frac{(x{:}\tau) \in \Gamma}{\Gamma \vdash_1 x} \qquad \frac{x, \Gamma \vdash_1 e}{\Gamma \vdash_1 \lambda x.\, e} \qquad \frac{(x{:}\tau), \Gamma \vdash_1 e : \mathsf{Any}}{\Gamma \vdash_1 \lambda(x{:}\tau).\, e} \qquad \frac{\Gamma \vdash_1 e : \lfloor \tau \rfloor}{\Gamma \vdash_1 \mathsf{stat}\ \tau\ e} \qquad \frac{\Gamma \vdash_1 e : \mathsf{Any}}{\Gamma \vdash_1 \mathsf{stat}\ e}$$

$\boxed{\Gamma \vdash_1 e : K}$

$$\frac{x \in \Gamma}{\Gamma \vdash_1 x : \mathsf{Any}} \qquad \frac{(x{:}\tau) \in \Gamma \quad \lfloor \tau \rfloor = K}{\Gamma \vdash_1 x : K} \qquad \frac{x, \Gamma \vdash_1 e}{\Gamma \vdash_1 \lambda x.\, e : \mathsf{Fun}} \qquad \frac{(x{:}\tau), \Gamma \vdash_1 e : \mathsf{Any}}{\Gamma \vdash_1 \lambda(x{:}\tau).\, e : \mathsf{Fun}}$$

$$\frac{\Gamma \vdash_1 e_0 : \mathsf{Any} \quad \Gamma \vdash_1 e_1 : \mathsf{Any}}{\Gamma \vdash_1 \langle e_0, e_1 \rangle : \mathsf{Pair}} \qquad \frac{i \in \mathbb{N}}{\Gamma \vdash_1 i : \mathsf{Nat}} \qquad \frac{}{\Gamma \vdash_1 i : \mathsf{Int}} \qquad \frac{\Gamma \vdash_1 e_0 : \mathsf{Fun} \quad \Gamma \vdash_1 e_1 : \mathsf{Any}}{\Gamma \vdash_1 e_0\ e_1 : \mathsf{Any}}$$

$$\frac{\Gamma \vdash_1 e : \mathsf{Pair}}{\Gamma \vdash_1 \mathsf{fst}\ e : \mathsf{Any}} \qquad \frac{\Gamma \vdash_1 e : \mathsf{Pair}}{\Gamma \vdash_1 \mathsf{snd}\ e : \mathsf{Any}} \qquad \frac{\Gamma \vdash_1 e_0 : K_0 \quad \Gamma \vdash_1 e_1 : K_1 \quad \Delta(op^2, K_0, K_1) = K}{\Gamma \vdash_1 op^2\ e_0\ e_1 : K} \qquad \frac{\Gamma \vdash_1 e : K' \quad K' \leqslant: K}{\Gamma \vdash_1 e : K} \qquad \frac{}{\Gamma \vdash_1 \mathsf{Err} : K}$$

$$\frac{\Gamma \vdash_1 e \quad \lfloor \tau \rfloor = K}{\Gamma \vdash_1 \mathsf{dyn}\ \tau\ e : K} \qquad \frac{\Gamma \vdash_1 e}{\Gamma \vdash_1 \mathsf{dyn}\ e : \mathsf{Any}} \qquad \frac{\Gamma \vdash_1 e : \mathsf{Any}}{\Gamma \vdash_1 \mathsf{chk}\ K\ e : K}$$

Fig. 9. Property judgments for the first-order embedding

a constructor-check to the body of every typed function. Using pseudo-syntax e_0; e_1 to represent sequencing, a suitable completion rule is:

$$\frac{(x{:}\tau), \Gamma \vdash e \leadsto e'}{\Gamma \vdash \lambda(x{:}\tau_d).\, e \leadsto \lambda(x{:}\tau_d).\, \mathsf{chk}\lfloor \tau_d \rfloor x;\ e'}$$

Lastly, the models do not mention union types, universal types, and recursive types—all of which are common tools for reasoning about dynamically-typed code. To extend the higher-order embedding with support for these types, the language must add new kinds of monitors to enforce type soundness for their elimination forms [72, 75]. To extend the first-order embedding, the language must add unions $K \cup K$ to its grammar of type constructor and must extend the $\lfloor \cdot \rfloor$ function. For a union type, let $\lfloor \tau_0 \cup \tau_1 \rfloor$ be $\lfloor \tau_0 \rfloor \cup \lfloor \tau_1 \rfloor$, i.e., the union of the constructors of its members. For a universal type $\forall \alpha.\, \tau$ let the constructor be $\lfloor \tau \rfloor$, and for a type variable let $\lfloor \alpha \rfloor$ be Any because there are no elimination forms for a universally-quantified type variable.[3] For a recursive type $\mu\alpha.\, \tau$, let the constructor be $\lfloor \tau[\alpha \leftarrow \mathsf{Empty}] \rfloor$ where Empty is an empty type.

3 PERFORMANCE

A performance comparison of the three approaches to migratory typing must use three distinct compilers for the same syntax and typing system. For the syntax and typing system, we use Typed Racket. For the compilers, we implement: the higher-order embedding using the Typed Racket

---

[3]This treatment of universal types fails to enforce parametricity.

compiler and optimizer; the erasure embedding by direct translation to (untyped) Racket; and the first-order embedding using a modified version of the Typed Racket compiler. This section presents the results of an *exhaustive* performance evaluation of the three compilers for ten functional (with mutable references) benchmark programs.

3.1 Implementation Overview

Typed Racket [80] is a migratory typing system for Racket that implements the higher-order embedding. As a full-fledged implementation, Typed Racket handles many more types than the language of figure 1 and supports (higher-order) casts so that developers can easily migrate a module even if the type system cannot cope with the programming idioms. Its run-time system guarantees that every boundary error attributes the fault to exactly one syntactic type boundary [82].

Removing the type annotations *and casts* from a Typed Racket program yields a valid Racket program. We use this transformation to compare the higher-order embedding to erasure.

To compare with the first-order approach, we modified the Typed Racket compiler to rewrite typed code and compile types to predicates that enforce type constructors. The implementation is available online; see the supplement for details [31].

The three approaches outlined above define three ways to compile a Typed Racket program to Racket: higher-order TR-H, erasure TR-E, and first-order TR-1. In the rest of this section, we reserve the name "Typed Racket" for the source language.

3.2 Method

The performance evaluation uses the exhaustive method for module-level migratory typing [33, 76]. Starting from a multi-module program, we migrate the whole program—ignoring any libraries outside the control of the normal user—to Typed Racket. From this fully-typed program, we generate all typed/untyped configurations by removing types from a subset of the modules. A program with N modules thus leads to 2^N configurations, a set that represents all the ways a developer might apply migratory typing to the untyped program for a fixed set of type annotations.

Since the promise of migratory typing is that a developer may choose to run any mixed-typed configuration, the main goal of the evaluation is to classify all configurations by their overhead relative to the completely-untyped configuration. The key measure is the number of D-deliverable configurations. A configuration is D-deliverable if it runs no more than Dx slower than the untyped configuation. If an implementation of migratory typing adds little overhead to mixed-typed programs, then a large percentage of its configurations are D-deliverable for a low value of D.

3.3 Protocol

The evaluation reports the performance of the higher-order (TR-H), erasure (TR-E), and first-order (TR-1) approaches on ten Typed Racket programs. Nine programs are the functional benchmarks from prior work on Typed Racket; the tenth is adapted from a JPEG library.[4]

For each configuration of each benchmark, and for both TR-H and TR-1, we collected a sequence of eight running times by running the program once to account for JIT warmup and then an additional eight times for the actual measurement. For TR-E we measured one sequence of running times because all configurations erase to the same program.

All measurements were collected sequentially using Racket v6.10.1 on an unloaded Linux machine with two physical AMD Opteron 6376 processors (a NUMA architecture) and 128GB RAM. The CPU cores on each processor ran at 2.30 GHz using the "performance" CPU governor.
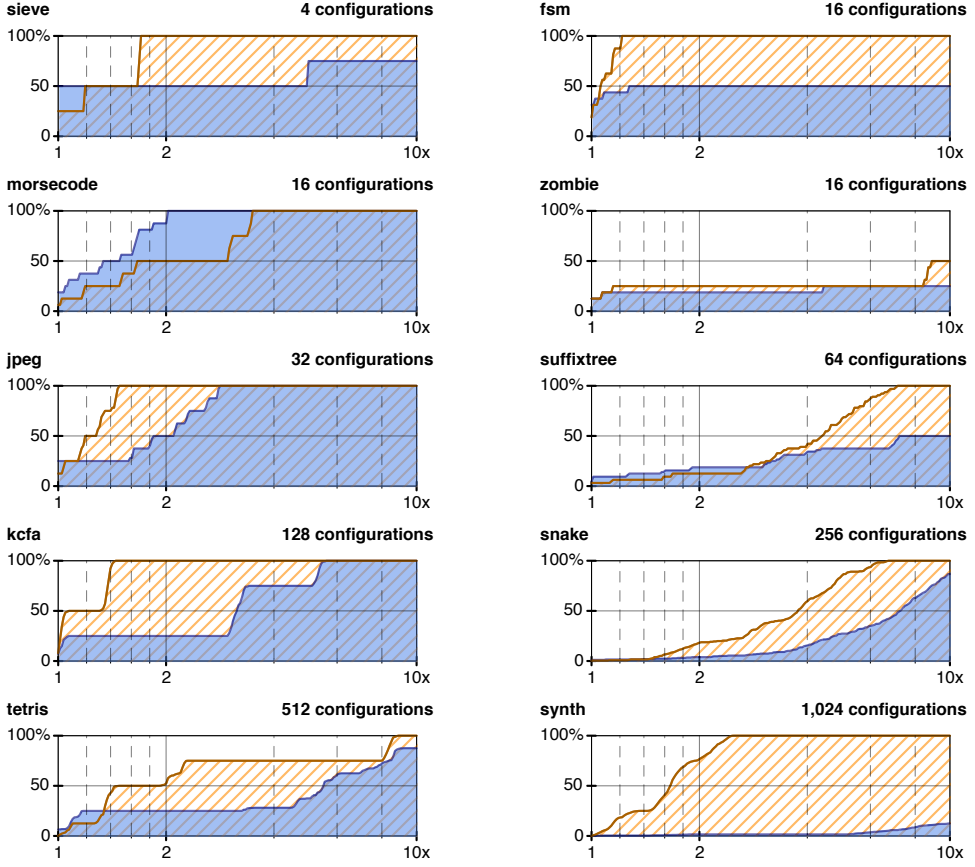

[4]docs.racket-lang.org/gtp-benchmarks

Fig. 10. TR-H (blue 🔵) and TR-1 (orange ⬭), each relative to erasure (TR-E). The *x*-axis is log-scaled. The unlabeled vertical ticks appear at: 1.2x, 1.4x, 1.6x, 1.8x, 4x, 6x, and 8x overhead. A larger area under the curve is better.

	sie.	fsm	mor.	zom.	jpeg	suf.	kcfa	sna.	tet.	syn.
TR-H	21.76x	506.10x	2.01x	1072.80x	2.81x	24.59x	5.57x	13.15x	13.93x	51.38x
TR-1	1.69x	1.21x	3.48x	20.36x	1.47x	7.10x	1.44x	6.72x	8.88x	2.49x

Fig. 11. Worst-case overhead for higher-order (TR-H) and first-order (TR-1), each relative to erasure.

3.4 Evaluation I: Mixed-Typed Programs

Figure 10 plots the overhead of TR-H relative to erasure (blue 🔵) and the overhead of TR-1 relative to erasure (orange ⬭) for the ten functional programs. The lines on each plot give the percent of *D*-deliverable configurations for values of *D* between 1 to 10. In other words, a point (X, Y) on a line for TR-H says that $Y\%$ of all TR-H configurations in this program run at most X times slower than the same program with all types erased.

Since seven of the ten benchmarks have at least one TR-H configuration that falls "off the charts" with an overhead above 10x, figure 11 tabulates the worst-case overhead in each benchmark. According to the table, the higher-order embedding may slow a working program by three orders
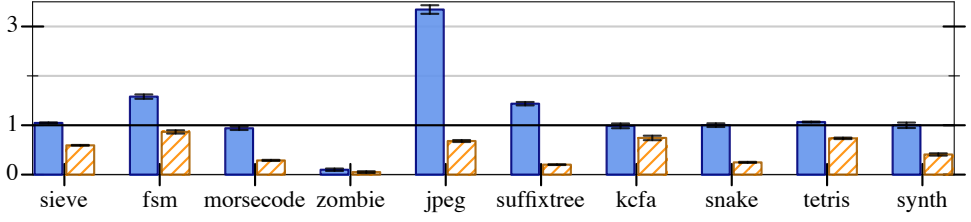
Fig. 12. Speedup of fully-typed TR-H (■) and TR-1 (▨), relative to TR-E (the 1x line). Taller bars are better than shorter bars.

of magnitude. The largest slowdowns, in `fsm` and `zombie`, occur because higher-order values repeatedly cross type boundaries and accumulate monitors. The worst-case performance of TR-1 is always within two orders of magnitude.

3.5 Evaluation II: Fully-Typed Programs

The table in figure 12 compares the performance of fully-typed programs (relative to libraries). The blue bars plot the overhead of TR-H relative to the erasure embedding on each benchmark. The orange bars plot analogous data for TR-1 relative to the erasure embedding.

The `jpeg` and `zombie` benchmarks are outliers. In `jpeg`, the speedup of TR-H over erasure is high because the user program depends on a typed library;[5] the library protects itself against TR-E code. In `zombie`, typed code is slower than erasure. The typed version of `zombie` performs a type cast in the inner loop. The untyped version replaces this cast with a rudimentary predicate check. This simple change noticeably affects the performance of the fully-typed configuration (the overhead of monitors, however, dominates the mixed-typed configurations).

3.6 Threats to Validity

The performance of a full-fledged TR-1 implementation may differ from that of our prototype.

On one hand, the prototype is likely to be faster than a full implementation because it makes no effort to provide useful error messages. When a constructor check fails, the prototype simply directs programmers to the source location of the check. Improving these error messages with information about the source of the incompatible value is likely to degrade performance in a significant manner.

On the other hand, the performance of a full implementation could improve over the prototype in two ways. First, TR-1 does not take advantage of the TR-H optimizer to remove checks for tag errors. Integrating the safe parts of the optimizer may offset some cost of the constructor checks. Second, the completion function for the prototype may introduce redundant checks. For example, consecutive reads from a list suffer the same check on the extracted element.

Three other threats are worth noting. First, TR-1 does not support Racket's object-oriented features [75]. We expect that scaling the implementation to the full language would not affect the functional benchmarks. Second, our benchmarks are relatively small; the largest is `jpeg` with approximately 1,500 lines of code. Third, the evaluation considers only one fully-typed version of each benchmark. Ascribing different types to the same program can affect its performance; for example, the check for an integer may run faster than the check for a natural number.

---

[5]To be clear, the TR-H, TR-E, and TR-1 versions of `jpeg` rely on the same typed library. We compile the library using TR-H in all cases because the original library is from Typed Racket and the original author of `jpeg` chose to use this library.

4   IMPLICATIONS

Sections 2 and 3 present the two critical aspects of the three approaches to combining statically typed and dynamically typed code via a twin pair of languages: (1) their semantics within a single framework and (2) their performance characteristics relative to a single base language on the same suite of benchmark programs. Equipped with this objective information, we can now explain the logical implications and the performance consequences of choosing one of these three approaches.

For the logical implications, we proceed in a type-directed manner. At the level of base types, there is no difference between the higher-order and first-order embeddings, but the erasure embedding may give a different result due to a violation of the types (section 4.1). After moving from base types to trees of base types, we can explain the truly essential difference between higher-order and first-order: while the higher-order embedding allows developers to reason compositionally about type annotations, users of the first-order variant must always consider the whole program (section 4.2). This non-compositional behavior means that a violation of the type annotations may go undetected in seemingly type-correct code. Higher-order types are similarly afflicted by the non-compositional behavior of the first-order embedding (section 4.3). Lastly, the three approaches provide radically different support when it comes to detecting, reporting, and debugging boundary errors (section 4.4).

For consequences with respect to performance, our work somewhat confirms the conjectures of the literature that lowering the standards of safety pays off—but only to some degree. While the first-order embedding adds less overhead than the higher-order embedding to a large portion of the mixed-typed programs (section 4.5), readers must keep two caveats in mind. For one, the first-order approach imposes a run-time checking overhead that is directly proportional to the number of types in the program. Second, the higher-order approach may exploit the full soundness of type annotations. As a result, programs with many type annotations tend to run faster under the higher-order semantics than the first-order one (section 4.6).

4.1   For Base Types

For a program that computes a value of base type, it can be tempting to think that dynamic typing (via erasure) provides all the soundness that matters in practice. After all, Ruby and Python throw a TypeError if a program attempts to add an integer to a string.

This claim is only true, however, if the static typing system is restricted to exactly match the host language's notion of dynamic typing. Adding a *logical* distinction between natural numbers and integers, as demonstrated in the type system of figure 2, can lead to silent failures at run-time when a negative integer flows into a context expecting a natural number. If the numbers represent votes, for example [82], then the lack of run-time checking can change the outcome of an election.

Other host languages may allow more diverse kinds of silent failures. JavaScript, for example, supports adding a number to a string, array, or object. TypeScript programmers must keep this behavior in mind, and may wish to use a library,[6] to protect their type-erased code against JavaScript.

Both the higher-order and first-order embeddings are sound for base types, e.g., if v is a value of type Nat, then v is a natural number. Informally, both embeddings fully-check base types.

4.2   For First-Order, Non-Base Types

The practical difference between the higher-order and first-order embeddings becomes clear in a mixed-typed program that deals with pairs. The higher-order embedding checks the contents of a pair; the first-order embedding only checks the constructor:[7]

---

[6]io.is is one such library: lorefnon.tech/2018/03/25/typescript-and-validations-at-runtime-boundaries
[7]In this and similar examples, we write $\vdash e : \tau \rightarrow^{*}_{1\text{-S}} e'$ to abbreviate: $\vdash e : \tau \rightsquigarrow e''$ for some e'' and $e'' \rightarrow^{*}_{1\text{-S}} e'$.

bessel.rkt

```
1 #lang typed
2
3 (define-type Bessel
4    (List Nonnegative-Real Real))
5
6 (: add-B (-> Bessel Bessel Bessel))
7 (define (add-B b0 b1)
8    (map + b0 b1))
```

student.rkt

```
1 #lang untyped
2
3 (require "bessel.rkt")
4
5 (define d0 (list 4 0))
6 (define d1 (list -2 1))
7
8 (add-B d0 d1)
```

Fig. 13. Logical error using polar-form complex numbers

$$\checkmark \quad \vdash \mathsf{dyn}\,(\mathsf{Nat}{\times}\mathsf{Nat})\,\langle -2, -2 \rangle : \mathsf{Nat}{\times}\mathsf{Nat} \rightarrow^{*}_{\mathsf{H\text{-}S}} \mathsf{BndryErr}$$

$$\triangle \quad \vdash \mathsf{dyn}\,(\mathsf{Nat}{\times}\mathsf{Nat})\,\langle -2, -2 \rangle : \mathsf{Nat}{\times}\mathsf{Nat} \rightarrow^{*}_{\mathsf{1\text{-}S}} \langle -2, -2 \rangle$$

Extracting a value from an ill-typed pair might not detect the mismatch, depending on what type of value the context expects. For example, a typed context can safely extract a negative integer from a pair of natural numbers if the context happens to expect an integer:

$$\checkmark \quad \vdash \mathsf{fst}\,(\mathsf{dyn}\,(\mathsf{Nat}{\times}\mathsf{Nat})\,\langle -2, -2 \rangle) : \mathsf{Nat} \rightarrow^{*}_{\mathsf{1\text{-}S}} \mathsf{BndryErr}$$

$$\triangle \quad \vdash \mathsf{fst}\,(\mathsf{dyn}\,(\mathsf{Nat}{\times}\mathsf{Nat})\,\langle -2, -2 \rangle) : \mathsf{Int} \rightarrow^{*}_{\mathsf{1\text{-}S}} -2$$

Similarly, a dynamically-typed expression can extract anything from a type-annotated pair:

$$\triangle \quad \vdash \mathsf{fst}\,(\mathsf{stat}\,(\mathsf{Nat}{\times}\mathsf{Nat})\,(\mathsf{dyn}\,(\mathsf{Nat}{\times}\mathsf{Nat})\,\langle -2, -2 \rangle)) \rightarrow^{*}_{\mathsf{1\text{-}D}} -2$$

Put another way, a developer cannot assume that a value of type $\tau_0{\times}\tau_1$ contains components of type τ_0 and type τ_1 because type-constructor soundness is not compositional.

Reynolds classic paper on types and abstraction begins with a similar example based on a distinction between real numbers and non-negative reals [60]:

> In one section, Professor Descartes announced that a complex number was an ordered pair of reals [...] In the other section, Professor Bessel announced that a complex number was an ordered pair of reals the first of which was nonnegative [...]

Figure 13 adapts this example to a mixed-typed world. The typed module on left defines addition for "Bessel-style" complex numbers; the function adds the components of the given pairs. The dynamically-typed module on the right mistakenly calls the addition function on two "Descartes-style" numbers, one of which does not match the type for Bessel numbers.

As it turns out, each of the three approaches to migratory typing behaves differently on this program. The higher-order embedding correctly rejects the application of add-B at the boundary:

$$\checkmark \quad \vdash (\mathsf{add\text{-}B}\ \mathsf{d0}\ \mathsf{d1}) \rightarrow_{\mathsf{H\text{-}S}} \mathsf{BndryErr}$$

The erasure embedding silently computes a well-typed, nonsensical result:

$$\triangle \quad \vdash (\mathsf{add\text{-}B}\ \mathsf{d0}\ \mathsf{d1}) \rightarrow^{*}_{\mathsf{E\text{-}S}} (\mathsf{list}\ 2\ 1)$$

The first-order embedding *either* computes a nonsensical result or raises a boundary error somewhere within the map function:

$$\triangle \quad \vdash (\mathsf{add\text{-}B}\ \mathsf{d0}\ \mathsf{d1}) \rightarrow^{*}_{\mathsf{1\text{-}S}} \begin{cases} (\mathsf{list}\ 2\ 1) & \text{if map does not check the Bessel type} \\ \mathsf{BndryErr} & \text{if map does check the type} \end{cases}$$

It is impossible to predict the outcome without knowing the local type annotations within map.

```
database.rkt                    typed-db.rkt                    app.rkt

1 #lang untyped            1 #lang typed                1 #lang untyped
2                          2                            2
3 (define (create db name) 3 (require/typed/provide      3 (require
4   (exec-query ...))      4   "database.rkt"            4   "typed-db.rkt")
5                          5   [#:opaque DB             5
6 ;; ...                   6    sql-connection?]        6 (define (serve r)
                           7   [create                 7   (if (new-user? r)
                           8    (-> DB Username         8       (create ...)
                           9        Boolean)])          9       ...))
                          10                           10
                          11 (define-type Username     11 ;; ...
                          12   Symbol)
```

Fig. 14. Adding types between two untyped modules

4.3 For Higher-Order Types

One promising application of migratory typing is to layer a typed interface over an existing, dynamically-typed library of functions. For the low effort of converting library documentation into a type specification, the library's author and clients benefit from a machine-checked API.

Figure 14 demonstrates this use-case. The module on the left represents a dynamically-typed library that manages a SQL database. The module on the right represents a dynamically-typed web application; the application uses the database library to create and access user accounts. In the middle, the type annotations formalize the interface between the database layer and the application.

With the higher-order embedding, a developer can trust the type annotations. The database module may assume well-typed arguments and the application is guaranteed well-typed results, despite the lack of static types within either module.

In contrast, the erasure embedding completely ignores types at run-time and treats the middle module of figure 14 as one large comment. The types are just for documentation and the IDE.

The first-order embedding provides a limited compromise: for every value that flows from untyped to typed, the implementation checks that the value constructor matches the type constructor. Concretely, there is one run-time check that ensures `create` is bound to a function.

This single check does little to verify the correctness of the dynamically-typed code. In terms of the model, retrofitting a "first-order" type onto a higher-order function f does not enforce that f respects its arguments:

$$f = (\lambda x.\, x\, \langle 1, 1 \rangle)$$
$$h = \text{dyn}\,(\text{Nat} \Rightarrow \text{Nat})\,(\lambda y.\, \text{sum}\, y\, y)$$
$$\vdash (\text{dyn}\,((\text{Nat} \Rightarrow \text{Nat}) \Rightarrow \text{Nat})\, f)\, h : \text{Nat} \rightarrow^*_{1\text{-S}} f\, h \rightarrow^*_{1\text{-S}} h\, \langle 1, 1 \rangle \rightarrow^*_{1\text{-S}} \text{TagErr}$$

Conversely, there is no guarantee that untyped clients of a function g abide by its interface:

$$g = \text{dyn}\,(\text{Int} \times \text{Int} \Rightarrow \text{Int})\,(\lambda x.\, \text{snd}\, x)$$
$$\vdash (\text{stat}\,(\text{Int} \Rightarrow \text{Int})\, g)\, 2 \rightarrow^*_{1\text{-D}} \text{snd}\, 2 \rightarrow^*_{1\text{-D}} \text{TagErr}$$

Thus the practical benefits of writing a typed API in a first-order system are vanishingly small.

4.4 For Errors and Error Messages

Error messages matter. As Vitousek et al. [83] claim, improved error messages are "one of the primary benefits" of adding types to a dynamically-typed language. To illustrate, they describe a

```
                                                    ┌──────────┐                              ┌───────────┐
                                                    │ stats.rkt │                              │ client.rkt │
                                                    └──────────┘                              └───────────┘
1 #lang typed                                                      1 #lang untyped
2 (require math)                                                   2
3                                                                  3 (define lst
4 (: moment (-> (Listof Float) Integer Float))                    4   (list "A" "B"))
5 (define (moment xs m)                                            5
6   (define u (mean xs))                                           6 (moment lst 2)
7   (define n (length xs))
8   (/ (sum (map (λ (x) (expt (- x u) m)) xs))
9      n))
```

Fig. 15. Type-mismatch between a library function and client, adapted from Vitousek et al. [83].

situation in which an untyped module sends a list of strings to a typed function that expects a list
of numbers:

> if the library authors make use of gradual typing [...] then the error can be localized and
> caught before the call to moment [...] the runtime error points to the call to moment

This claim assumes, of course, that the gradual typing system enforces types.

Figure 15 turns their illustration into a concrete example. The stats module computes the m-th
moment of a list of floats; it defers most of the computation to the language's list package, meaning
a call to moment may cause another boundary-crossing. The client module calls moment with an
inappropriate list.

The higher-order embedding catches the error before the call to moment:

$$\checkmark \quad \vdash (\text{moment lst 2}) \rightarrow^*_{\text{H-D}} (\text{moment } (\mathcal{D}_H(\text{Listof}(\text{Float}), \text{lst})) \text{ 2}) \rightarrow^*_{\text{H-D}} \text{BndryErr}$$

The erasure embedding performs the call to moment just like a dynamically-typed language would.
If the numeric operations check their inputs, the execution ends in a tag error. If the primitives are
un-checked, however, then the call may compute a nonsensical result:

$$\triangle \quad \vdash (\text{moment lst 2}) \rightarrow^*_{\text{E-D}} \begin{cases} \text{TagErr} & \text{if mean or } - \text{ check for strings} \\ -42 & \text{if the primitives are unchecked} \end{cases}$$

The first-order embedding confirms that lst is a list and then proceeds with the call. Since the
body of moment never directly extracts a float from the list, it is impossible to predict what happens
during the call. For example, mean can raise a boundary error, raise a tag error, or silently compute
a sum of string pointers:

$$\triangle \quad \vdash (\text{moment lst 2}) \rightarrow^*_{\text{1-D}} \begin{cases} \text{BndryErr} & \text{if mean, } -, \text{ or map use the @ttFloat type} \\ \text{TagErr} & \text{if mean or } - \text{ check for strings} \\ -42 & \text{if the primitives are unchecked} \end{cases}$$

In the case of a boundary error, it is not clear how a first-order embedding can pinpoint the boundary
that is violated. Vitousek et al. [84] propose a strategy that points the first-order error message to
the call to moment, but the strategy may double the running time of a program and reports a set of
potentially-guilty boundaries rather than pinpointing the faulty one.

By contrast, the higher-order embedding can identify the first violation of the types — even for
higher-order interactions — by storing debugging information in monitor values [82]. With the
relevant boundary term, the developer knows exactly where to begin debugging: either the type
annotation is wrong or the dynamically-typed code does not match the type.

Generally speaking, higher-order discovers more errors than first-order and first-order discovers
more errors than erasure:

Theorem 2.9 : Err *approximation*
If $e \in e_S$ and $\vdash e : \tau$ then the following statements hold:
- if $e \rightarrow^*_{\text{E-S}} \text{Err}$ then $e \rightarrow^*_{\text{1-S}} \text{Err}$
- if $e \rightarrow^*_{\text{1-S}} \text{Err}$ then $e \rightarrow^*_{\text{H-S}} \text{Err}$

Proof (sketch): Informally, each pair of reduction relations is "equivalent" up to their strategies for enforcing static types. The supplement defines these equivalences as simulation relations [31]. □

The reverse implications do not hold. As section 4.1 explains, the expression (dyn Nat −2) steps to a boundary error via $\rightarrow^*_{\text{1-S}}$ but not via the $\rightarrow^*_{\text{E-S}}$ relation. Section 4.2 presents an example where $\rightarrow^*_{\text{H-S}}$ steps a boundary error and $\rightarrow^*_{\text{1-S}}$ produces a value.

4.5 For the Performance of Mixed-Typed Programs

Enforcing soundness in a mixed-typed program adds performance overhead. As the graphs in section 3 demonstrate, this cost can be high (10x) in the first-order embedding and enormous (1000x) in the higher-order embedding.

The first-order embedding incurs type-constructor checks at three places: type boundaries, applications of typed functions, and explicit chk terms. While each check adds a small cost,[8] these costs accumulate. The added code and branches may affect JIT compilation.

The higher-order embedding incurs three significant kinds of costs. First, there is the cost of checking a value at a boundary. Second, there is an allocation cost when a higher-order value crosses a boundary. Third, monitored values suffer an indirection cost; for example, a monitor guarding a dynamically-typed function must check every result computed by the function.

Each kind of cost may be arbitrarily large. The (time) cost of checking an algebraic type depends on the size of the given value. The (time and space) cost of allocation grows with the number of boundary-crossings, as does the (time) cost of indirection. In the following example, an untyped function crosses three boundaries and consequently accumulates three monitors:

$$\begin{aligned} &\text{dyn} (\text{Nat} \Rightarrow \text{Nat}) (\text{stat} (\text{Int} \Rightarrow \text{Int}) (\text{dyn} (\text{Int} \Rightarrow \text{Int}) \, \lambda x. \, x)) \\ &\rightarrow^*_{\text{H-S}} \text{mon} (\text{Nat} \Rightarrow \text{Nat}) (\text{mon} (\text{Int} \Rightarrow \text{Int}) (\text{mon} (\text{Int} \Rightarrow \text{Int}) \, \lambda x. \, x)) \end{aligned}$$

Finally, the indirection added by monitors may limit the effectiveness of a JIT compiler.

4.6 For the Performance of Fully-Typed Programs

If a program has few dynamically-typed components, then the first-order embedding is likely to perform the worst of the three embeddings. This poor performance comes about because all typed expressions unconditionally check their inputs. For example, a function that adds both elements of a pair value must check that its input has integer-valued components:

$$\begin{aligned} &\vdash \lambda(x : \text{Int} \times \text{Int}). \, \text{sum} \, (\text{fst} \, x) \, (\text{snd} \, x) : \text{Int} \times \text{Int} \Rightarrow \text{Int} \\ &\rightsquigarrow \lambda(x : \text{Int} \times \text{Int}). \, \text{sum} \, (\text{chk Int} \, (\text{fst} \, x)) \, (\text{chk Int} \, (\text{snd} \, x)) \end{aligned}$$

As a rule-of-thumb, adding types imposes (at least) a linear-time performance degradation [31, 32].

The higher-order embedding pays to enforce soundness only if static and dynamic components interact. If there are few interactions, the program spends little time enforcing soundness.

Furthermore, the soundness of the higher-order embedding means that a compiler can apply classic, type-directed optimizations. Thus the higher-order embedding's performance can exceed that of the erasure embedding, as shown in figure 12. Typed Racket (TR-H) in particular applies optimizations to unbox primitive values, select low-level primitive operations, provide fast access to data structures, and eliminate unused branches [69, 70].

---

[8]In the model, checks have $O(1)$ cost. In the implementation, checks have near-constant cost $O(n)$ where n is the number of types in the widest union type ($\tau_0 \cup \ldots \cup \tau_{n-1}$) in the program.

Higher-Order Embedding

Gradualtalk[†] [4],
TPD[†] [86], Typed Racket[†] [80]   StrongScript [62]

Erasure Embedding

ActionScript[†] [57], mypy[†],
Flow[†] [17], Hack[†], Pyre[†], Pytype[†],
Pyret[†],   rtc[†] [59], Strongtalk[†] [16],
Thorn [89]   TypeScript[†] [12], Typed Clojure[†] [15],
Typed Lua[†] [43]

First-Order Embedding

Dart 2, Nom [51], Reticulated[†] [84],
SafeTS [58], TR-1[†] (section 3)   († : migratory typing system)
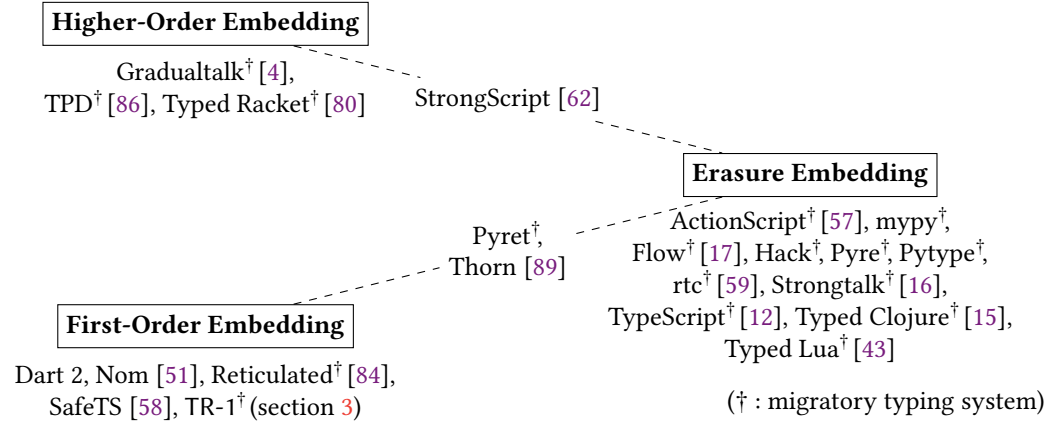
Fig. 16. Design space of migratory and mixed-typed systems.

5 EXISTING SYSTEMS

Figure 16 classifies existing migratory and mixed-typed systems in terms of the three approaches.[9] Systems listed under the box labeled *higher-order embedding* enforce higher-order types at run-time. Systems under the *erasure embedding* label provide an optional static type checker but do not use types to determine program behavior. Systems under the *first-order embedding* label enforce type boundaries with some form of first-order checks — the details vary between systems. In Dart 2 and Nom, every structured value is associated with run-time type information (e.g., the value is an object and is associated with a class name); the boundary checks perform a subtype test using this type information. SafeTS is similar, however, the type information is structural rather than nominal and may gain new fields (but not methods) by crossing a boundary. Reticulated and our TR-1 prototype perform first-order checks similar to those outlined in section 2.5 and furthermore rewrite statically-typed code to protect against untyped values.

Several systems are located on dashed lines in figure 16 because they compromise between two approaches. StrongScript and Thorn include two kinds of types: concrete types and like types. Both types are checked statically, but only concrete types are enforced at run-time. In other words, a program that uses only like types has erasure behavior. These two related systems are on different lines because only StrongScript supports higher-order types (such types must be concrete).

Pyret falls between the first-order and erasure approaches. If a program contains type annotations, then Pyret enforces each annotation with a run-time type constructor check. A programmer can therefore opt into type-constructor soundness through disciplined use of type annotations.

6 RELATED WORK

The idea of equipping a dynamically typed language with static type information goes back at least to the compiler hints in MACLISP [50]. Early work focused on type reconstruction for dynamically-typed programs [7, 73, 87]. Over the past decade, researchers turned to the problem of creating a multi-language system [29] that provides a type soundness guarantee [40, 46, 66, 79].

---

[9]The interested reader may wish to consult the supplement, in which we instantiate the framework of section 2 for several existing systems [31]. The supplement also has URLs for the languages.

6.1 Gradual Typing

Migratory typing is closely related to gradual typing [66, 67]. In the broad sense, the term gradual typing has come to describe any type system that allows some amount of dynamic typing. In the precise sense of Siek et al. [67], 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 type with the dynamic type can only (3a) remove a compile-time type error or (3b) remove a run-time boundary error.

Gradual typing and migratory typing have different goals. Migratory typing always starts with a dynamically typed language, whereas gradual typing may begin with a static type system and add a dynamic type [20, 28, 41], an idea that also goes back decades [1, 42, 78].

6.2 Concrete Types

Thorn is a statically-typed language that allows dynamically-typed methods [14, 89]. In particular: every value in Thorn is an instance of a class; every value has a (concrete) type, i.e., the name of its class; and a method may be defined for a dynamically-typed argument, in which case the method uses a run-time subtype check before interacting with its argument. This approach sacrifices expressiveness in favor of straightforward run-time checks. Richards et al. [62] apply the concrete approach to TypeScript and allow limited interaction with structurally-typed JavaScript objects; method calls are permitted, but typed and JavaScript objects cannot extend one another.[10] Muehlboeck and Tate [51] develop a theory of concrete and gradual [67] typing and present an efficient implementation. Dynamic typing in Dart 2 is based on the concrete approach.[11]

6.3 Higher-Order, Erasure, and First-Order

Matthews and Findler [46] use the name *natural embedding* to describe a type-directed strategy of converting between Scheme and ML values. Their name suggests that this inductive-checking, higher-order-wrapping technique is the obvious approach to the problem; indeed, work on typed foreign-function interfaces [56] and remote procedure calls [53] used a similar approach. New and Licata [52] provide a semantic justification for the name; in brief, an embedding is unsound if it allows untyped functions but is not equivalent to the *natural* wrapping strategy.

The erasure approach is better known as optional typing, and the idea dates back to Common Lisp [71] and Strongtalk [16]. Many languages now have optional type checkers (figure 16).

The first-order embedding presented in section 2.5 is directly inspired by the transient semantics for Reticulated Python [83, 84]. Transient begins with an uninterpreted surface language expression and elaborates it into a typed intermediate language with explicit type-constructor checks. The main judgment has the form $\Gamma \vdash e \rightsquigarrow e' : \tau$ where both e' and τ are outputs.

Henglein [36] uses the name *completion process* to describe a procedure that adds type-constructor checks to the syntax of an untyped expression. Both Henglein's completion process and our completion function are examples of type-directed coercion insertions [9, 74].

6.4 Type Reconstruction

While the erasure approach converts typed code to untyped code, a *reconstruction embedding* could convert all untyped code to typed code. Researchers have worked on variants of this problem for decades. Soft typing combines Hindley-Milner inference with a non-standard grammar of types [3, 87]. Set-based flow analysis infers a type based on values, primitive operations, and control-flow [26, 34, 35, 47, 54, 57]. Still another method is to infer types from the completion of an untyped term;

---

[10]Takikawa et al. [77] introduce *opaque class contracts* to support mixed-typed class hierarchies in Typed Racket.
[11]dartlang.org/guides/language/sound-dart, accessed 2018-05-10

that is, from a term with all implicit constructor-checks made explicit [37]. In practice there are two major challenges for type reconstruction: the known algorithms are not suitable for large programs [48] and their inference is syntactically brittle [82].

6.5 Performance of Mixed-Typed Programs

Herman et al. [38] recognize the problem of space-inefficiency in the higher-order embedding and propose a theoretical solution. Other theoretical solutions address the issue for gradual typing [38, 65, 68], and more generally for higher-order contracts [30].

Recent work evaluates the performance of practical migratory typing systems. Allende et al. [6] report the performance of mixed-typed Gradualtalk programs. Takikawa et al. [76] introduce a systematic method for performance evaluation and report a high overhead for mixed-typed programs in Typed Racket. Bauman et al. [11] demonstrate that a tracing JIT compiler can significantly reduce the overhead in Typed Racket. In the related space of concrete types, Muehlboeck and Tate [51] report excellent performance for a new gradually-typed language. Richards et al. [61] suggest integrating run-time type checks with the shape tests of an optimizing virtual machine.

6.6 Type Soundness

At least four prior works address aspects of mixed-typed soundness. The early work on Typed Racket [79] explains why soundness for a pair of languages requires a more general theorem than soundness for a single language. A `like` type system [62, 89] allows the programmer to decide between enforced and erased types. Confined gradual typing [5] offers a choice between a static type error and a run-time check in the higher-order approach. Lastly, progressive types [55] describes a type system with a tunable set of run-time errors.[12]

6.7 Blame

Correct blame is an important consolation prize because a migratory typing system cannot guarantee the absence of certain run-time errors the way a statically-typed language can. Correct blame helps with debugging such errors by attributing the fault to one specific type boundary [23].

Typed Racket informally guarantees blame correctness [82]. Muehlboeck and Tate [51] formally prove an *immediate accountability* property that implies blame correctness, albeit for a language that limits the expressiveness of untyped code.

The first publication on Typed Racket (which pre-dates the first formal statements of *correct blame* [23] and *complete monitoring* [24]) states that the evaluation of a typed expression cannot end in a tag error [79]. Wadler and Findler [85] adapt this property to a type system with a dynamic type and name it "the blame theorem." Unlike correct blame, this less-precise property has been adapted to many other systems [2, 39, 64, 67, 84].

6.8 Comparing Gradual Typing Systems

Siek et al. [63] define three calculi for gradual typing and relate them with fully-abstract translations. The three calculi provide identical soundness guarantees.

Chung et al. [19] study the relationship of four different designs of object-oriented gradual typing without inheritance. The paper presents a core language, dubbed KafKa, which is implemented in .NET and provably type-sound. The comparison rests on four translations from the surface syntax to KafKa, each of which formulates a different semantics of gradual typing. Finally, the paper compares the four approaches with examples, showing how the resulting behaviors differ.

---

[12]By contrast, this paper makes an argument for "preservation-ive types".

7 FINDING BALANCE

This paper contributes two major results. First, it delivers a theoretical framework for investigating different ways of combining twin pairs of dynamically-typed and statically-typed languages. The framework generalizes the Matthews–Findler multi-language approach [46] and the theorems in section 2 clearly show how soundness for a pair of languages requires a more careful treatment than soundness for a single language. With the framework, we can finally work out a systematic comparison of prior work *and* capture the first-order semantics of Reticulated [84] in such a way that it is easy to create the first alternative implementation.

Second, this paper is the first to present an apples-to-apples performance evaluation of three implementations of these primary semantics of migratory typing. This evaluation weakly confirms conjectures in the literature, which is valuable, but most importantly it shows that none of these approaches dominates across the whole spectrum. Jointly the two contributions put the systematic and comparative study of a spectrum on a firm basis that allows well-founded conclusions.

In practice, each approach has different implications for how a developer can reason about the code, especially when diagnosing the cause of a run-time error:

- Running a TR-E (erasure) program gives a developer no clue as to what in the source code triggers an error; i.e., the type information in the code does *not* reduce the search space. Indeed, a violation of the types in the source code may go completely unnoticed.
- Running a TR-1 (first-order) program is guaranteed to reveal a violation of types *if it affects the execution of typed code.* The checking schema is unlikely to pinpoint the source of the error, however.
- Running a TR-H (higher-order) program uncovers a violation of type annotations as soon as there is a witness and pinpoints the exact type boundary that is violated by this witness.

One open question is whether developers want correctness and precise run-time errors.

In terms of performance, the picture is more nuanced than the literature suggests. On a mixed-typed program: erasure adds no overhead, first-order checks add overhead on a pay-as-you-annotate basis, and the higher-order approach may render a working program unusably slow. For fully-typed programs, the higher-order embedding often provides the best performance of all three. Equipped with this comparison platform, we intend to explore additional ways of making some form of sound migratory typing sufficiently practical for software development.

One strategy is to improve the completion function and evaluation property of first-order model; the pair in section 2.5 is correct, but simplistic. Occurrence typing [81] seems well-suited for this task. A second strategy is to design a JIT compiler that can dynamically minimize the cost of run-time constructor checks; the HiggsCheck compiler [61] might be a promising context in which to experiment. Alternatively, combining the first-order approach with the Pycket [10, 11] JIT compiler for Racket may yield an implementation with good performance in all configurations. A third strategy is to combine multiple semantics.

A APPENDIX

This appendix presents two alternative higher-order approaches, called *co-natural* and *forgetful*, and their logical implications. Co-natural enforces all non-base types with monitors [21, 25]. Forgetful limits each value to at most one monitor [30]. Full definitions and proofs are in the supplement [31].

One might also explore an approach that monitors base values and further delays errors [21, 22].

A.1 Co-Natural Embedding

Figure 19 presents the co-natural embedding. Its evaluation syntax extends the surface syntax with monitors for functions and pairs. The $\mathcal{D}_C$ boundary function checks that an untyped value matches the expected type constructor and wraps all function and pair values in a monitor. Likewise, $\mathcal{S}_C$ wraps functions and pairs. The reduction rules in figure 19 specify the behavior of monitored values. Soundness for the co-natural embedding states that reduction preserves the property in figure 20.

A.2 Forgetful Embedding

The forgetful embedding, defined in figure 17, prevents a value from accumulating more than one monitor. If a monitored value reaches one of the $\mathcal{D}_F$ or $\mathcal{S}_F$ boundary functions, the function replaces the existing monitor. Consequently, a statically-typed function that crosses two type boundaries may be wrapped in a monitor with an incompatible type; let $f = (\lambda(x\!:\!\mathsf{Int}).\, -2)$ in:

$$\vdash \mathsf{dyn}\,(\mathsf{Nat}\!\Rightarrow\!\mathsf{Nat})\,(\mathsf{stat}\,(\mathsf{Int}\!\Rightarrow\!\mathsf{Int})\,f) : \mathsf{Nat}\!\Rightarrow\!\mathsf{Nat} \rightarrow^*_{\mathsf{F\text{-}S}} \mathsf{mon}\,(\mathsf{Nat}\!\Rightarrow\!\mathsf{Nat})\,f$$

The evaluation syntax therefore includes the $\mathsf{chk}\ \tau\ e$ expression to check the result of a monitored, typed function against the type declared in its monitor. Soundness for the forgetful embedding states that reduction preserves the property in figure 18.

A.3 Implications of Co-Natural and Forgetful

The co-natural approach delays run-time checks until the relevant part of a value is accessed. Thus a type error can go undiscovered if it does not affect the particular execution:

$$\triangle \quad \vdash \mathsf{fst}\,(\mathsf{dyn}\,(\mathsf{Nat}\!\times\!\mathsf{Nat})\,\langle 2, -2\rangle) : \mathsf{Nat}\!\times\!\mathsf{Nat} \rightarrow^*_{\mathsf{C\text{-}S}} 2$$

Unlike the first-order approach, however, co-natural can find such errors in untyped contexts as well as typed contexts, thus preventing the miscalculation demonstrated in section 4:

$$\checkmark \quad \vdash \mathsf{snd}\,(\mathsf{stat}\,(\mathsf{Nat}\!\times\!\mathsf{Nat})\,(\mathsf{dyn}\,(\mathsf{Nat}\!\times\!\mathsf{Nat})\,\langle 2, -2\rangle)) \rightarrow^*_{\mathsf{C\text{-}D}} \mathsf{BndryErr}$$

$$\triangle \quad \vdash \mathsf{snd}\,(\mathsf{stat}\,(\mathsf{Nat}\!\times\!\mathsf{Nat})\,(\mathsf{dyn}\,(\mathsf{Nat}\!\times\!\mathsf{Nat})\,\langle 2, -2\rangle)) \rightarrow^*_{\mathsf{1\text{-}D}} -2$$

The forgetful approach can also detect errors in untyped contexts:

$$\checkmark \quad \vdash \mathsf{snd}\,(\mathsf{stat}\,(\mathsf{Nat}\!\times\!\mathsf{Nat})\,(\mathsf{dyn}\,(\mathsf{Nat}\!\times\!\mathsf{Nat})\,\langle 2, -2\rangle)) \rightarrow^*_{\mathsf{F\text{-}D}} \mathsf{BndryErr}$$

Co-natural and forgetful differ in their approach to pairs (or functions) that cross multiple boundary terms. In the following example, an untyped pair flows in and out of typed code. The first type annotation does not match the value and the forgetful approach fails to detect the mismatch:

$$\checkmark \quad \vdash \mathsf{snd}\,(\mathsf{stat}\,(\mathsf{Int}\!\times\!\mathsf{Int})\,(\mathsf{dyn}\,(\mathsf{Nat}\!\times\!\mathsf{Nat})\,\langle 2, -2\rangle)) \rightarrow^*_{\mathsf{C\text{-}D}} \mathsf{BndryErr}$$

$$\triangle \quad \vdash \mathsf{snd}\,(\mathsf{stat}\,(\mathsf{Int}\!\times\!\mathsf{Int})\,(\mathsf{dyn}\,(\mathsf{Nat}\!\times\!\mathsf{Nat})\,\langle 2, -2\rangle)) \rightarrow^*_{\mathsf{F\text{-}D}} -2$$

Unlike the first-order embedding, the run-time checks in the forgetful embedding come from boundary terms, not from the client context:

$$\checkmark \quad \vdash \mathsf{snd}\,(\mathsf{dyn}\,(\mathsf{Nat}\!\times\!\mathsf{Nat})\,\langle 2, -2\rangle) : \mathsf{Int} \rightarrow^*_{\mathsf{F\text{-}S}} \mathsf{BndryErr}$$

$$\triangle \quad \vdash \mathsf{snd}\,(\mathsf{dyn}\,(\mathsf{Nat}\!\times\!\mathsf{Nat})\,\langle 2, -2\rangle) : \mathsf{Int} \rightarrow^*_{\mathsf{1\text{-}S}} -2$$

$\boxed{\text{Language HF}}$ extends Evaluation Syntax

$$
\begin{array}{rcl}
e & = & \dots \mid \mathsf{chk}\,\tau\,e \\
v & = & \dots \mid \mathsf{mon}\,(\tau \Rightarrow \tau)\,(\lambda x.\,e) \mid \mathsf{mon}\,(\tau \Rightarrow \tau)\,(\lambda(x\!:\!\tau).\,e) \mid \mathsf{mon}\,(\tau \times \tau)\,\langle v, v\rangle \\
E^\bullet & = & \dots \mid \mathsf{chk}\,\tau\,E^\bullet \\
E & = & \dots \mid \mathsf{chk}\,\tau\,E
\end{array}
$$

$\boxed{\mathcal{D}_\mathsf{F} : \tau \times v \longrightarrow e}$ $\boxed{\mathcal{S}_\mathsf{F} : \tau \times v \longrightarrow e}$

$\mathcal{D}_\mathsf{F}(\tau, v) = \mathcal{X}(\tau, v)$ $\mathcal{S}_\mathsf{F}(\tau, v) = \mathcal{X}(\tau, v)$

$\boxed{\mathcal{X} : \tau \times v \longrightarrow e}$

$$
\begin{array}{lll}
\mathcal{X}(\tau_d \Rightarrow \tau_c, \lambda x.\,e) & = & \mathsf{mon}\,(\tau_d \Rightarrow \tau_c)\,(\lambda x.\,e) \\
\mathcal{X}(\tau_d \Rightarrow \tau_c, \lambda(x\!:\!\tau).\,e) & = & \mathsf{mon}\,(\tau_d \Rightarrow \tau_c)\,(\lambda(x\!:\!\tau).\,e) \\
\mathcal{X}(\tau_d \Rightarrow \tau_c, \mathsf{mon}\,(\tau'_d \Rightarrow \tau'_c)\,v') & = & \mathsf{mon}\,(\tau_d \Rightarrow \tau_c)\,v' \\
\mathcal{X}(\tau_0 \times \tau_1, \langle v_0, v_1\rangle) & = & \mathsf{mon}\,(\tau_0 \times \tau_1)\,\langle v_0, v_1\rangle \\
\mathcal{X}(\tau_0 \times \tau_1, \mathsf{mon}\,(\tau'_0 \times \tau'_1)\,v') & = & \mathsf{mon}\,(\tau_0 \times \tau_1)\,v' \\
\mathcal{X}(\mathsf{Int}, i) & = & i \\
\mathcal{X}(\mathsf{Nat}, i) & = & i \\
\quad \text{if } i \in \mathbb{N} & & \\
\mathcal{X}(\tau, v) & = & \mathsf{BndryErr} \\
\quad \text{otherwise} & &
\end{array}
$$

$\boxed{e \rhd_{\mathsf{S\text{-}1}} e}$ extends $\rhd_\mathsf{S}$ $\boxed{e \rhd_{\mathsf{D\text{-}1}} e}$ extends $\rhd_\mathsf{D}$

$$
\begin{array}{ll}
\mathsf{dyn}\,\tau\,v & \rhd_{\mathsf{S\text{-}1}} \mathcal{D}_\mathsf{F}(\tau, v) \\
\mathsf{chk}\,\tau\,v & \rhd_{\mathsf{S\text{-}1}} \mathcal{X}(\tau, v) \\
(\mathsf{mon}\,(\tau_d \Rightarrow \tau_c)\,(\lambda x.\,e))\,v & \rhd_{\mathsf{S\text{-}1}} \mathsf{dyn}\,\tau_c\,e' \\
\quad \text{where } e' = (\lambda x.\,e)\,(\mathcal{X}(\tau_d, v)) \\
(\mathsf{mon}\,(\tau_d \Rightarrow \tau_c)\,(\lambda(x\!:\!\tau).\,e))\,v \rhd_{\mathsf{S\text{-}1}} \mathsf{chk}\,\tau_c\,e' \\
\quad \text{where } e' = (\lambda(x\!:\!\tau).\,e)\,(\mathcal{X}(\tau, v)) \\
\mathsf{fst}\,(\mathsf{mon}\,(\tau_0 \times \tau_1)\,\langle v_0, v_1\rangle) & \rhd_{\mathsf{S\text{-}1}} \mathcal{X}(\tau_0, v_0) \\
\mathsf{snd}\,(\mathsf{mon}\,(\tau_0 \times \tau_1)\,\langle v_0, v_1\rangle) & \rhd_{\mathsf{S\text{-}1}} \mathcal{X}(\tau_1, v_1)
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{stat}\,\tau\,v & \rhd_{\mathsf{D\text{-}1}} \mathcal{S}_\mathsf{F}(\tau, v) \\
& \\
(\mathsf{mon}\,(\tau_d \Rightarrow \tau_c)\,(\lambda x.\,e))\,v & \rhd_{\mathsf{D\text{-}1}} (\lambda x.\,e)\,v \\
& \\
(\mathsf{mon}\,(\tau_d \Rightarrow \tau_c)\,(\lambda(x\!:\!\tau).\,e))\,v \rhd_{\mathsf{D\text{-}1}} \mathsf{stat}\,\tau_c\,e' \\
\quad \text{where } e' = \mathsf{chk}\,\tau_c\,((\lambda(x\!:\!\tau).\,e)\,(\mathcal{X}(\tau, v))) \\
\mathsf{fst}\,(\mathsf{mon}\,(\tau_0 \times \tau_1)\,\langle v_0, v_1\rangle) & \rhd_{\mathsf{D\text{-}1}} \mathcal{X}(\tau_0, v_0) \\
\mathsf{snd}\,(\mathsf{mon}\,(\tau_0 \times \tau_1)\,\langle v_0, v_1\rangle) & \rhd_{\mathsf{D\text{-}1}} \mathcal{X}(\tau_1, v_1)
\end{array}
$$

$\boxed{e \rightarrow^*_{\mathsf{F\text{-}S}} e}$ similar to $\rightarrow^*_{\mathsf{H\text{-}S}}$, see tech. rpt. $\boxed{e \rightarrow^*_{\mathsf{F\text{-}D}} e}$ similar to $\rightarrow^*_{\mathsf{H\text{-}D}}$, see tech. rpt.

Fig. 17. Forgetful Embedding

$\boxed{\Gamma \vdash_\mathsf{F} e}$ extends $\Gamma \vdash e$ (selected rules)

$$
\dfrac{\Gamma \vdash_\mathsf{F} v_0 \quad \Gamma \vdash_\mathsf{F} v_1}{\Gamma \vdash_\mathsf{F} \mathsf{mon}\,(\tau_0 \times \tau_1)\,\langle v_0, v_1\rangle} \qquad
\dfrac{\Gamma \vdash_\mathsf{F} v_0 : \tau'_0 \quad \Gamma \vdash_\mathsf{F} v_1 : \tau'_1}{\Gamma \vdash_\mathsf{F} \mathsf{mon}\,(\tau_0 \times \tau_1)\,\langle v_0, v_1\rangle}
$$

$\boxed{\Gamma \vdash_\mathsf{F} e : \tau}$ extends $\Gamma \vdash e : \tau$ (selected rules)

$$
\dfrac{\Gamma \vdash_\mathsf{F} e : \tau'}{\Gamma \vdash_\mathsf{F} \mathsf{chk}\,\tau\,e : \tau} \qquad
\dfrac{\Gamma \vdash_\mathsf{F} \lambda x.\,e}{\Gamma \vdash_\mathsf{F} \mathsf{mon}\,(\tau_d \Rightarrow \tau_c)\,\lambda x.\,e : (\tau_d \Rightarrow \tau_c)} \qquad
\dfrac{\Gamma \vdash_\mathsf{F} \lambda(x\!:\!\tau'_d).\,e : \tau'_d \Rightarrow \tau'_c}{\Gamma \vdash_\mathsf{F} \mathsf{mon}\,(\tau_d \Rightarrow \tau_c)\,\lambda(x\!:\!\tau'_d).\,e : (\tau_d \Rightarrow \tau_c)}
$$

Fig. 18. Property judgments for the forgetful embedding

$\boxed{\text{Language HC}}$ extends Evaluation Syntax

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

$\boxed{\mathcal{D}_{\mathsf{C}} : \tau \times v \longrightarrow e}$

$\mathcal{D}_{\mathsf{C}}(\tau_d \Rightarrow \tau_c, v) = \mathsf{mon}\,(\tau_d \Rightarrow \tau_c)\,v$
  if $v = \lambda x.\,e$ or $v = \mathsf{mon}\,(\tau'_d \Rightarrow \tau'_c)\,v'$
$\mathcal{D}_{\mathsf{C}}(\tau_0 \times \tau_1, v) \ = \mathsf{mon}\,(\tau_0 \times \tau_1)\,v$
  if $v = \langle v_0, v_1 \rangle$ or $v = \mathsf{mon}\,(\tau'_0 \times \tau'_1)\,v'$
$\mathcal{D}_{\mathsf{C}}(\mathsf{Int}, i) \quad = i$
$\mathcal{D}_{\mathsf{C}}(\mathsf{Nat}, i) \quad = i$
  if $i \in \mathbb{N}$
$\mathcal{D}_{\mathsf{C}}(\tau, v) \qquad = \mathsf{BndryErr}$
  otherwise

$\boxed{\mathcal{S}_{\mathsf{C}} : \tau \times v \longrightarrow e}$

$\mathcal{S}_{\mathsf{C}}(\tau_d \Rightarrow \tau_c, v) = \mathsf{mon}\,(\tau_d \Rightarrow \tau_c)\,v$

$\mathcal{S}_{\mathsf{C}}(\tau_0 \times \tau_1, v) \ = \mathsf{mon}\,(\tau_0 \times \tau_1)\,v$

$\mathcal{S}_{\mathsf{C}}(\tau, v) \qquad = v$
  otherwise

$\boxed{e \triangleright_{\mathsf{S\text{-}C}} e}$ extends $\triangleright_{\mathsf{S}}$

$\mathsf{dyn}\,\tau\,v \qquad\qquad\qquad \triangleright_{\mathsf{S\text{-}C}} \ \mathcal{D}_{\mathsf{C}}(\tau, v)$
$(\mathsf{mon}\,(\tau_d \Rightarrow \tau_c)\,v_f)\,v \ \triangleright_{\mathsf{S\text{-}C}} \ \mathsf{dyn}\,\tau_c\,(v_f\,e')$
  where $e' = \mathsf{stat}\,\tau_d\,v$
$\mathsf{fst}\,(\mathsf{mon}\,(\tau_0 \times \tau_1)\,v) \quad \triangleright_{\mathsf{S\text{-}C}} \ \mathsf{dyn}\,\tau_0\,(\mathsf{fst}\,v)$
$\mathsf{snd}\,(\mathsf{mon}\,(\tau_0 \times \tau_1)\,v) \ \triangleright_{\mathsf{S\text{-}C}} \ \mathsf{dyn}\,\tau_1\,(\mathsf{snd}\,v)$

$\boxed{e \triangleright_{\mathsf{D\text{-}C}} e}$ extends $\triangleright_{\mathsf{D}}$

$\mathsf{stat}\,\tau\,v \qquad\qquad\qquad \triangleright_{\mathsf{D\text{-}C}} \ \mathcal{S}_{\mathsf{C}}(\tau, v)$
$(\mathsf{mon}\,(\tau_d \Rightarrow \tau_c)\,v_f)\,v \ \triangleright_{\mathsf{D\text{-}C}} \ \mathsf{stat}\,\tau_c\,(v_f\,e')$
  where $e' = \mathsf{dyn}\,\tau_d\,v$
$\mathsf{fst}\,(\mathsf{mon}\,(\tau_0 \times \tau_1)\,v) \quad \triangleright_{\mathsf{D\text{-}C}} \ \mathsf{stat}\,\tau_0\,(\mathsf{fst}\,v)$
$\mathsf{snd}\,(\mathsf{mon}\,(\tau_0 \times \tau_1)\,v) \ \triangleright_{\mathsf{D\text{-}C}} \ \mathsf{stat}\,\tau_1\,(\mathsf{snd}\,v)$

$\boxed{e \rightarrow^{*}_{\mathsf{C\text{-}S}} e}$ similar to $\rightarrow^{*}_{\mathsf{H\text{-}S}}$, see tech. rpt.

$\boxed{e \rightarrow^{*}_{\mathsf{C\text{-}D}} e}$ similar to $\rightarrow^{*}_{\mathsf{H\text{-}D}}$, see tech. rpt.

Fig. 19. Co-Natural Embedding

$\boxed{\Gamma \vdash_{\mathsf{C}} e}$ extends $\Gamma \vdash e$

$$\frac{\Gamma \vdash_{\mathsf{C}} v : \tau_0 \times \tau_1}{\Gamma \vdash_{\mathsf{C}} \mathsf{mon}\,(\tau_0 \times \tau_1)\,v} \qquad \frac{\Gamma \vdash_{\mathsf{C}} v : \tau_d \Rightarrow \tau_c}{\Gamma \vdash_{\mathsf{C}} \mathsf{mon}\,(\tau_d \Rightarrow \tau_c)\,v}$$

$\boxed{\Gamma \vdash_{\mathsf{C}} e : \tau}$ extends $\Gamma \vdash e : \tau$

$$\frac{\Gamma \vdash_{\mathsf{C}} v}{\Gamma \vdash_{\mathsf{C}} \mathsf{mon}\,(\tau_0 \times \tau_1)\,v : (\tau_0 \times \tau_1)} \qquad \frac{\Gamma \vdash_{\mathsf{C}} v}{\Gamma \vdash_{\mathsf{C}} \mathsf{mon}\,(\tau_d \Rightarrow \tau_c)\,v : (\tau_d \Rightarrow \tau_c)}$$

Fig. 20. Property judgments for the co-natural embedding

ACKNOWLEDGMENTS

REFERENCES

[1] Martin Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. Dynamic Typing in a Statically Typed Language. *Transactions on Programming Languages and Systems* 13(2), pp. 237–268, 1991.

[2] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for All. *Symposium on Principles of Programming Languages*, pp. 201–214, 2011.

[3] Alexander Aiken, Edward L. Wimmers, and T.K. Lakshman. Soft Typing with Conditional Types. *Symposium on Principles of Programming Languages*, pp. 163–173, 1994.

[4] 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.

[5] Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. Confined Gradual Typing. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 251–270, 2014.

[6] Esteban Allende, Johan Fabry, and Éric Tanter. Cast Insertion Strategies for Gradually-Typed Objects. *Dynamic Languages Symposium*, pp. 27–36, 2013.

[7] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards Type Inference for JavaScript. *European Conference on Object-Oriented Programming*, pp. 428–452, 2005.

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

[9] Gilles Barthe. Implicit Coercions in Type Systems. *International Workshop on Types for Proofs and Programs*, pp. 1–15, 1995.

[10] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. Pycket: A Tracing JIT For a Functional Language. *International Conference on Functional Programming*, pp. 22–34, 2015.

[11] Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. Sound Gradual Typing: Only Mostly Dead. *Proceedings of the ACM on Programming Languages* 1(OOPSLA), pp. 54:1–54:24, 2017.

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

[13] Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding Dynamic Types to C#. *European Conference on Object-Oriented Programming*, pp. 76–100, 2010.

[14] 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. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 117–136, 2009.

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

[16] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 215–230, 1993.

[17] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levy. Fast and Precise Type Checking for JavaScript. *Proceedings of the ACM on Programming Languages* 1(OOPSLA), pp. 48:1–48:30, 2017.

[18] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent Types for JavaScript. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 587–606, 2012.

[19] Benjamin W. Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. KafKa: Gradual Typing for Objects. *European Conference on Object-Oriented Programming*, pp. 12:1–12:23, 2018.

[20] Matteo Cimini and Jeremy G. Siek. The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems. *Symposium on Principles of Programming Languages*, pp. 443–455, 2016.

[21] Markus Degen, Peter Thiemann, and Stefan Wehr. The Interaction of Contracts and Laziness. *Workshop on Partial Evaluation and Program Manipulation*, pp. 97–106, 2012.

[22] Christos Dimoulas and Matthias Felleisen. On Contract Satisfaction in a Higher-Order World. *Transactions on Programming Languages and Systems* 33(5), pp. 16:1–16:29, 2011.

[23] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct Blame for Contracts: No More Scapegoating. *Symposium on Principles of Programming Languages*, pp. 215–226, 2011.

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

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

[26] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching Bugs in the Web of Program Invariants. *Conference on Programming Language Design and Implementation*, pp. 23–32, 1996.

[27] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static Type Inference for Ruby. *Symposium on Applied Computing*, pp. 1859–1866, 2009.

[28] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting Gradual Typing. *Symposium on Principles of Programming Languages*, pp. 429–442, 2016.

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

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

[31] Ben Greenman and Matthias Felleisen. A Spectrum of Type Soundness and Performance: Supplementary Material. Northeastern University, NU-CCIS-2018-002, 2018.

[32] Ben Greenman and Zeina Migeed. On the Cost of Type-Tag Soundness. *Workshop on Partial Evaluation and Program Manipulation*, pp. 30–39, 2018.

[33] 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, 2016.

[34] Nevin Heintze. Set-based analysis of ML-programs. *LISP and Functional Programming*, pp. 306–317, 1994.

[35] Thomas S. Heinze, Anders Møller, and Fabio Strucco. Type Safety Analysis for Dart. *Dynamic Languages Symposium*, pp. 1–12, 2016.

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

[37] Fritz Henglein and Jakob Rehof. Safe Polymorphic Type Inference for a Dynamically Typed Language: Translating Scheme to ML. *International Conference on Functional Programming Languages and Computer Architecture*, pp. 192–203, 1995.

[38] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient Gradual Typing. *Higher-Order and Symbolic Computation* 23(2), pp. 167–189, 2010.

[39] Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. On Polymorphic Gradual Typing. *Proceedings of the ACM on Programming Languages* 1(ICFP), pp. 40:1–40:29, 2017.

[40] Kenneth Knowles and Cormac Flanagan. Hybrid Type Checking. *Transactions on Programming Languages and Systems* 32(6), pp. 1–34, 2010.

[41] Nico Lehmann and Éric Tanter. Gradual Refinement Types. *Symposium on Principles of Programming Languages*, pp. 775–788, 2017.

[42] Xavier Leroy and Michael Mauny. Dynamics in ML. *International Conference on Functional Programming Languages and Computer Architecture*, pp. 406–426, 1991.

[43] Andre Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. A Formalization of Typed Lua. *Dynamic Languages Symposium*, pp. 13–25, 2015.

[44] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. Typed Lua: An Optional Type System for Lua. *Workshop on Dynamic Languages and Applications*, pp. 1–10, 2014.

[45] Simon Marlow and Philip Wadler. A Practical Subtyping System for Erlang. *International Conference on Functional Programming*, pp. 136–149, 1997.

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

[47] Philippe Meunier, Robert Bruce Findler, and Matthias Felleisen. Modular Set-Based Analysis from Contracts. *Symposium on Principles of Programming Languages*, pp. 218–231, 2006.

[48] Philippe Meunier, Robert Bruce Findler, Paul Steckler, and Mitchell Wand. Selectors Make Set-Based Analysis Too Hard. *Higher-Order and Symbolic Computation* 18, pp. 245–269, 2005.

[49] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences* 17(3), pp. 348–375, 1978.

[50] David A. Moon. MACLISP Reference Manual. MIT, Revision 0, 1974.

[51] Fabian Muehlboeck and Ross Tate. Sound Gradual Typing is Nominally Alive and Well. *Proceedings of the ACM on Programming Languages* 1(OOPSLA), pp. 56:1–56:30, 2017.

[52] Max S. New and Daniel R. Licata. Call-by-name Gradual Type Theory. *International Conference on Formal Structures for Computation and Deduction*, 2018.

[53] Atsushi Ohori and Kazuhiko Kato. Semantics for Communication Primitives in a Polymorphic Language. *Symposium on Principles of Programming Languages*, pp. 99–112, 1993.

[54] Frédéric Pluquet, Antoine Marot, and Roel Wuyts. Fast Type Reconstruction for Dynamically Typed Programming Languages. *Dynamic Languages Symposium*, pp. 69–78, 2009.

[55] Joe Gibbs Politz, Hannah Quay-de la Vallee, and Shriram Krishnamurthi. Progressive Types. *Onward!*, pp. 55–66, 2012.

[56] Norman Ramsey. Embedding an interpreted language using higher-order functions and types. *Journal Functional Programming* 21(6), pp. 585–615, 2008.

[57] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The Ins and Outs of Gradual Type Inference. *Symposium on Principles of Programming Languages*, pp. 481–494, 2012.

[58] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. *Symposium on Principles of Programming Languages*, pp. 167–180, 2015.

[59] Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. The Ruby Type Checker. *Symposium on Applied Computing*, pp. 1565–1572, 2013.

[60] John C. Reynolds. Types, Abstraction, and Parametric Polymorphism. *Information Processing*, pp. 513–523, 1983.

[61] Gregor Richards, Ellen Arteca, and Alexi Turcotte. The VM Already Knew That: Leveraging Compile-Time Knowledge to Optimize Gradual Typing. *Proceedings of the ACM on Programming Languages* 1(OOPSLA), pp. 55:1–55:27, 2017.

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

[63] Jeremy Siek, Peter Thiemann, and Philip Wadler. Blame and Coercion: Together Again for the First Time. *Conference on Programming Language Design and Implementation*, pp. 425–435, 2015.

[64] Jeremy Siek, Michael M. Vitousek, Matteo Cimmini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic References for Efficient Gradual Typing. *European Symposium on Programming*, pp. 432–456, 2015.

[65] Jeremy G. Siek, Ronald Garcia, and Walid Taha. Exploring the Design Space of Higher-Order Casts. *European Symposium on Programming*, pp. 17–31, 2009.

[66] Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. *Scheme and Functional Programming*. University of Chicago, TR-2006-06, 2006.

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

[68] Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. *Symposium on Principles of Programming Languages*, pp. 365–376, 2010.

[69] Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. Optimization coaching. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 163–178, 2012.

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

[71] Guy L. Steele. *Common Lisp the Language*. 2nd edition. Digital Press, 1990.

[72] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and Impersonators: Run-time Support for Reasonable Interposition. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 943–962, 2012.

[73] Norihisa Suzuki. Inferring Types in Smalltalk. *Symposium on Principles of Programming Languages*, pp. 187–199, 1981.

[74] Nikhil Swamy, Michael Hicks, and Gavin M. Bierman. A Theory of Typed Coercions and its Applications. *International Conference on Functional Programming*, pp. 329–340, 2009.

[75] Asumu Takikawa, Daniel Feltey, Earl Dean, Robert Bruce Findler, Matthew Flatt, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards Practical Gradual Typing. *European Conference on Object-Oriented Programming*, pp. 4–27, 2015.

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

[77] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual Typing for First-Class Classes. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 793–810, 2012.

[78] Satish Thatte. Quasi-static Typing. *Symposium on Principles of Programming Languages*, pp. 367–381, 1990.

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

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

[81] Sam Tobin-Hochstadt and Matthias Felleisen. Logical Types for Untyped Languages. *International Conference on Functional Programming*, pp. 117–128, 2010.

[82] 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. *Summit oN Advances in Programming Languages*, pp. 17:1–17:17, 2017.

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

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

[85] Philip Wadler and Robert Bruce Findler. Well-typed Programs Can't be Blamed. *European Symposium on Programming*, pp. 1–15, 2009.

[86] Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. Mixed Messages: Measuring Conformance and Non-Interference in TypeScript. *European Conference on Object-Oriented Programming*, pp. 28:1–28:29, 2017.

[87] Andrew K. Wright and Robert Cartwright. A Practical Soft Type System for Scheme. *Transactions on Programming Languages and Systems* 19(1), pp. 87–152, 1997.

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

[89] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating Typed and Untyped Code in a Scripting Language. *Symposium on Principles of Programming Languages*, pp. 377–388, 2010.