Deep and Shallow Types for Gradual Languages

Ben Greenman*
Brown University
USA
benjamin.l.greenman@gmail.com

Abstract

Sound gradual types come in many forms and offer varying levels of soundness. Two extremes are deep types and shallow types. Deep types offer compositional guarantees but depend on expensive higher-order contracts. Shallow types enforce only local properties, but can be implemented with first-order checks. This paper presents a language design that supports both deep and shallow types to utilize their complementary strengths.

In the mixed language, deep types satisfy a strong complete monitoring guarantee and shallow types satisfy a first-order notion of type soundness. The design serves as the blueprint for an implementation in which programmers can easily switch between deep and shallow to leverage their distinct advantages. On the GTP benchmark suite, the median worst-case overhead drops from several orders of magnitude down to 3x relative to untyped. Where an exhaustive search is feasible, 40% of all configurations run fastest with a mix of deep and shallow types.

CCS Concepts: • **Software and its engineering → Semantics**; Constraints; Functional languages.

Keywords: gradual typing, migratory typing, complete monitoring, type-enforcement strategies

ACM Reference Format:
Ben Greenman. 2022. Deep and Shallow Types for Gradual Languages. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/3519939.3523430

1 A Spectrum of Type Enforcement

Taken broadly, the research area of gradual typing presents several *type-enforcement strategies* that enforce static types against untyped code to varying levels of fidelity. Among strategies that are compatible with an untyped host language and provide a type soundness guarantee, two promising alternatives are Natural [40, 60, 52] and Transient [70]. The Natural strategy uses higher-order contracts to enforce the behavioral claims implied by higher-order types. The Transient strategy uses first-order checks to enforce basic aspects of types. Unsurprisingly, these two strategies come with different benefits and drawbacks. Contracts in Natural enable *deep* types that satisfy type soundness and complete monitoring [27]. These contracts, however, can impose a huge performance cost [30]. First-order checks in Transient enable only *shallow* types, which promise a weak soundness guarantee, but these checks rarely dominate the running time of a program [70, 29, 51].

The question thus arises as to whether the two enforcement strategies can interoperate, giving programmers deep types when guarantees matter and shallow types to avoid performance bottlenecks. This paper provides an affirmative answer via three contributions.

- A theoretical model that integrates deep-typed code, shallow-typed code, and untyped code via a semantics that applies ideas from Natural and Transient (section 3). The model comes with two essential metatheorems: the first validates plain type soundness for shallow-typed code, and the second shows that deep-typed code retains the customary type soundness property via complete monitoring.
- An implementation of Typed Racket [63] that permits developers to combine deep, shallow, and untyped components (section 4). The deep and shallow halves of the implementation stand on equal footing. Switching between them is a one-line change.
- A practical evaluation of the performance, guarantees, and expressiveness of the revised Typed Racket implementation (section 5). The performance study of this novel three-way Typed Racket demonstrates significant improvements on the GTP benchmark suite [32] over the two-way versions.

Deep and shallow types can interoperate without sacrificing their formal properties. Best of all, the combination brings measurable benefits. These contributions strongly suggest that combining type-sound gradual typing strategies is an

[*] Research done at Northeastern University

PLDI '22, June 13–17, 2022, San Diego, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523430

effective means to give programmers control over the protection/performance tradeoff.

2 Background

2.1 Gradual, Migratory, Mixed-Typed

Gradual typing explores combinations of static and dynamic typing [52, 60, 40, 31]. The goal of this research is a language that supports two styles of code in a convenient manner. Untyped code is free to perform any computation that the language can express. Typed code is restricted to well-typed computations, but comes with a guarantee that static types are meaningful predictions about run-time behaviors. Differences among gradual languages arise over what makes for a convenient mix. True gradually-typed languages include a universal `Dynamic` type that helps to blur the distinction between typed and untyped code [54]. Migratory typing systems add idiomatic types to an existing language [63]. Other mixed-typed methods include the development of novel languages [72, 44, 38, 51] and compilers [49, 5].

With these various end-goals in mind, our formal development (section 3) begins with two restrictions: types may only be enforced with ahead-of-time techniques and there is no dynamic type. These rules ensure a widely-applicable baseline for languages that can mix typed and untyped code.

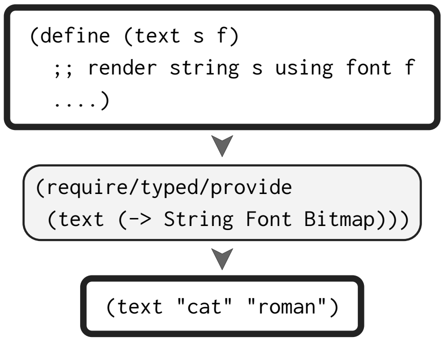
2.2 Deep and Shallow Types

Sound gradual language designs do not agree on how types should guide the behavior of a program. Two leading alternatives for run-time properties are deep and shallow types. To a first approximation, deep types enforce (but do not verify) the same guarantees as conventional static types and shallow types enforce only local type soundness.

Figure 1 presents a three-module program to illustrate the gap between deep and shallow types. The untyped module on top contains a stub definition for a function `text` that expects two arguments. This module is a simplified picture of the Racket `images/icons/symbol` module, which incorporates thousands of lines of rendering and raytracing code—a module that is easiest left untyped. The typed module in the middle is an interface for the untyped function, which passes on (in a higher-order manner) to clients who might rely on the type. The type correctly states that `text` expects a string and a font object and computes a bitmap object. Finally, the untyped client module on the bottom mistakenly calls `text` with two strings instead of one string and one object.

The question raised by this example is whether static types can catch the mistake in the untyped client. Deep and shallow types give opposite answers:

- Deep types enforce the typed interface with run-time obligations for both the client and the library. Because the client sends a string where the type expects a font object, the client triggers a run-time type error.



```
(define (text s f)
  ;; render string s using font f
  ....)
```

```
(require/typed/provide
 (text (-> String Font Bitmap)))
```

```
(text "cat" "roman")
```

Figure 1: Untyped library, typed interface, and untyped client

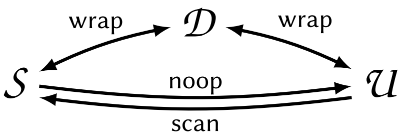


Figure 2: Outline for deep, shallow, and untyped interactions

- Shallow types guarantee the local integrity of typed code, but nothing more. The untyped client is allowed to send any input to the untyped `text` function, including two strings, without causing a type-level error.

From a theoretical perspective, shallow types satisfy a type soundness property and nothing more [70, 26]. Soundness states that the type of an expression predicts the kinds of values that evaluation can produce. In typed code, these predictions are often specific and useful. For example, an expression with a function type cannot evaluate to a number. In untyped code, these predictions are trivial; soundness merely ensures a well-formed result. A property that distinguishes deep types from shallow is complete monitoring [13, 27]. Semantics that satisfy complete monitoring enforce types as invariants that all clients, typed or untyped, can rely on.

2.2.1 Natural Semantics. One way to implement deep types is the Natural semantics [60, 40, 52].[1] Natural interprets types as contracts in a straightforward manner.[2] For example, base types are enforced with predicate checks, types for immutable values are enforced with first-order traversals, and types for higher-order values such as arrays and functions are enforced with higher-order *wrapper* contracts. Because each contract fully enforces a type, these contracts need only guard the boundaries between typed and untyped code. Within typed modules, code can run efficiently and employ type-directed optimizations [59].

[1] Natural is a.k.a. Guarded [68], Behavioral [9], and Deep [65].

[2] Researchers are actively seeking improved variants of Natural [35, 22, 55, 53, 21] and measuring the efficiency of implementations [15, 38]. Theoretical results about Natural hold for these semantics-preserving variants as well.

$$\begin{aligned} s &= x \mid i \mid \langle s, s\rangle \mid \lambda x.\, s \mid \lambda(x\!:\!\tau).\, s \mid \lambda(x\!:\!\underline{\tau}).\, s \mid \\ &\quad \mathit{unop}\ s \mid \mathit{binop}\ s\ s \mid \mathsf{app}\ s\ s \mid \mathsf{module}\, L\ s \\ L &= \mathcal{D} \mid \mathcal{S} \mid \mathcal{U} \\ \tau &= \mathsf{Nat} \mid \mathsf{Int} \mid \tau\times\tau \mid \tau\rightarrow\tau \\ T &= \tau \mid \underline{\tau} \mid \mathcal{U} \end{aligned}$$

Figure 3: Surface syntax

2.2.2 Transient Semantics. The Transient semantics is an implementation of shallow types that does not require wrappers [70]. Transient enforces types by injecting first-order checks throughout typed pieces of code: typed, public functions must check their inputs; typed modules must check their untyped imports; and typed expressions must check the results computed during a function call, the elements extracted from a data structure, and the outcome of any downcasts. In figure 2, these conditions imply one check: the typed interface must check that `text` is a function. In general, every line of typed code may add several Transient checks, but each check is inexpensive. By contrast to higher-order contracts, the checks do not traverse values and do not impose allocation and indirection costs.

3 Model and Metatheory

A normal gradual language allows for two styles of code, typed and untyped, and uses run-time checks to enforce the claims made by static types. Our model allows for three syntaxes: deep-typed code, shallow-typed code, and untyped code. Both deep and shallow code must satisfy the same type checker, which validates conventional well-formedness properties. Untyped code has fewer constraints. Run-time checks enforce type claims at boundaries, but use different strategies for deep and for shallow types.

Overall, the primary goal of the model is to test whether deep, shallow, and untyped code can safely interoperate. A secondary goal of the model is to outline an implementation. For this reason, the three syntaxes compile to one kernel language that can express a variety of standard run-time checks: a *wrap* term applies a contract, a *scan* term performs a first-order (predicate) check, and a *noop* term represents a boundary that any value may cross. Figure 2 sketches the plan for applying these terms at type boundaries in a way that protects deep and shallow code from untyped values (including values that have passed through a typed context).

3.1 Three-Way Surface Syntax

The surface syntax (figure 3) equips a basic expression language with optional type annotations and module boundaries. Surface expressions s consist of function applications ($\mathsf{app}\, s\, s$), primitive operation applications ($\mathit{unop}\, s$, $\mathit{binop}\, s\, s$), variables x, integers i, pairs $\langle s, s\rangle$, and optionally-annotated functions. An untyped function has no annotation ($\lambda x.\, s$), a deep-typed function has a plain type annotation ($\lambda(x\!:\!\tau).\, s$), and a shallow-typed function has an underlined type annotation ($\lambda(x\!:\!\underline{\tau}).\, s$). The underline is a syntactic hint that only the top-level shape of this type is guaranteed at runtime. Types τ express natural numbers (Nat), integers (Int), pairs ($\tau\times\tau$), and functions ($\tau\rightarrow\tau$). Modules associate a label with an expression ($\mathsf{module}\, L\ s$). The label L is either $\mathcal{D}$ for deep-typed code, $\mathcal{S}$ for shallow-typed code, or $\mathcal{U}$ for untyped code. For example, the term ($\mathsf{module}\ \mathcal{D}\ s_0$) says that s_0 is a deep-typed expression. Any module expressions within s_0 are free to use any typing style ($\mathcal{D}$, $\mathcal{S}$, or $\mathcal{U}$).

3.2 Three-Way Surface Typing

Deep and shallow code must satisfy strong (and equal) type constraints. Untyped code is subject to a weaker constraint; namely, it cannot reference variables that it did not bind. These well-formedness conditions are spelled out in the typing judgment of figure 4, which relates a type environment Γ and an expression s to a result specification. A result T is either a type τ for deep-typed code, an underlined type $\underline{\tau}$ for shallow-typed code, or the uni-type $\mathcal{U}$ for untyped code.

With the exception of modules, the typing rules are standard for a basic functional language. Modules allow any kind of expression to appear within another. For instance, an untyped expression may appear within a deep expression provided that the untyped code is well-formed. There are seven such rules to ensure that the module language (L_0) matches the type of the subexpression (T_0); figure 4 presents these rules as one template rule (in [brackets]) and a table.

Figure 4 also defines a subtyping judgment (<:) and a type-assignment for primitive operations (Δ). Subtyping declares that the natural numbers are a subset of the integers and extends this covariantly to pairs and contra/co-variantly to function domains/codomains. The primitive operations are consequently overloaded to accept natural numbers or integers. The purpose of this basic subtyping judgment is not to sketch out a numeric tower [3], but rather to show that an upcast (via subtyping) can weaken the run-time checks that shallow code inserts. Weakening may have serious pragmatic implications for gradual union, intersection, and object types [8, 61, 2, 58, 36, 64].

3.3 Common Evaluation Syntax

Evaluation expressions e consist of variables, values, primitive applications, function applications, errors, and boundary terms. Unlike the surface syntax, there are no module terms. Instead, the three *boundary terms* describe run-time checks. A wrap boundary asks for the full enforcement of a type, either with a comprehensive first-order check or a higher-order wrapper. A scan boundary asks for a first-order type-shape (σ) check. A noop boundary asks for no check.

Values and errors represent the possible results of an evaluation. The values are integers, pairs, functions, and guard wrappers. A guard wrapper ($\mathbb{G}\ (\tau_0 \rightarrow \tau_1)\ v_0$) provides type-restricted access to a function. Shallow-typed functions have

$\boxed{\Gamma \vdash_{\mathsf{s}} s : T}$

$$\frac{(x_0 : \tau_0) \in \Gamma}{\Gamma \vdash_{\mathsf{s}} x_0 : \tau_0} \qquad \frac{(x_0 : \lfloor \tau_0 \rfloor) \in \Gamma}{\Gamma \vdash_{\mathsf{s}} x_0 : \lfloor \tau_0 \rfloor} \qquad \frac{(x_0 : \mathcal{U}) \in \Gamma}{\Gamma \vdash_{\mathsf{s}} x_0 : \mathcal{U}}$$

$$\frac{(x_0 : \tau_0), \Gamma \vdash_{\mathsf{s}} e_0 : \tau_1}{\Gamma \vdash_{\mathsf{s}} \lambda(x_0 : \tau_0).\, e_0 : \tau_0 \rightarrow \tau_1} \qquad \frac{(x_0 : \lfloor \tau_0 \rfloor), \Gamma \vdash_{\mathsf{s}} e_0 : \lfloor \tau_1 \rfloor}{\Gamma \vdash_{\mathsf{s}} \lambda(x_0 : \lfloor \tau_0 \rfloor).\, e_0 : \lfloor \tau_0 \rightarrow \tau_1 \rfloor}$$

$$\frac{(x_0 : \mathcal{U}), \Gamma \vdash_{\mathsf{s}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathsf{s}} \lambda x_0.\, e_0 : \mathcal{U}} \qquad \frac{\Gamma \vdash_{\mathsf{s}} e_0 : \lfloor \tau_0 \rightarrow \tau_1 \rfloor \quad \Gamma \vdash_{\mathsf{s}} e_1 : \lfloor \tau_0 \rfloor}{\Gamma \vdash_{\mathsf{s}} \mathsf{app}\, e_0\, e_1 : \lfloor \tau_1 \rfloor}$$

$$\frac{\Gamma \vdash_{\mathsf{s}} e_0 : \tau_0 \quad \tau_0 <: \tau_1}{\Gamma \vdash_{\mathsf{s}} e_0 : \tau_1} \qquad \frac{\Gamma \vdash_{\mathsf{s}} e_0 : \lfloor \tau_0 \rfloor \quad \tau_0 <: \tau_1}{\Gamma \vdash_{\mathsf{s}} e_0 : \lfloor \tau_1 \rfloor}$$

$$\left[\frac{\Gamma \vdash_{\mathsf{s}} e_0 : T_0}{\Gamma \vdash_{\mathsf{s}} \mathsf{module}\, L_0\, e_0 : T_1} \right]$$

L_0	T_0	T_1
$\mathcal{D}$	τ_0	τ_0
$\mathcal{D}$	τ_0	$\lfloor \tau_0 \rfloor$
$\mathcal{D}$	τ_0	$\mathcal{U}$
$\mathcal{S}$	$\lfloor \tau_0 \rfloor$	τ_0
$\mathcal{S}$	$\lfloor \tau_0 \rfloor$	$\lfloor \tau_0 \rfloor$
$\mathcal{S}$	$\lfloor \tau_0 \rfloor$	$\mathcal{U}$
$\mathcal{U}$	$\mathcal{U}$	T_1

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

$$\frac{}{\mathsf{Nat} <: \mathsf{Int}} \qquad \frac{\tau_0 <: \tau_2 \quad \tau_1 <: \tau_3}{\tau_0 \times \tau_1 <: \tau_2 \times \tau_3} \qquad \frac{\tau_2 <: \tau_0 \quad \tau_1 <: \tau_3}{\tau_0 \rightarrow \tau_1 <: \tau_2 \rightarrow \tau_3}$$

$\boxed{\Delta : \mathit{unop} \times \tau \longrightarrow \tau}$

$$\begin{aligned} \Delta(\mathsf{fst}, \tau_0 \times \tau_1) &= \tau_0 \\ \Delta(\mathsf{snd}, \tau_0 \times \tau_1) &= \tau_1 \end{aligned}$$

$\boxed{\Delta : \mathit{binop} \times \tau \times \tau \longrightarrow \tau}$

$$\begin{aligned} \Delta(\mathsf{plus}, \mathsf{Nat}, \mathsf{Nat}) &= \mathsf{Nat} \\ \Delta(\mathsf{plus}, \mathsf{Int}, \mathsf{Int}) &= \mathsf{Int} \\ \Delta(\mathsf{quotient}, \mathsf{Nat}, \mathsf{Nat}) &= \mathsf{Nat} \\ \Delta(\mathsf{quotient}, \mathsf{Int}, \mathsf{Int}) &= \mathsf{Int} \end{aligned}$$

Figure 4: Surface typing (selected rules), subtyping, and types for primitive operations

$$\begin{aligned} e \quad &= x \mid v \mid \langle e, e \rangle \mid \mathit{unop}\, e \mid \mathit{binop}\, e\, e \mid \mathsf{app}\, e\, e \mid \\ &\quad\ \mathsf{Error} \mid \mathsf{wrap}\, \tau\, e \mid \mathsf{scan}\, \sigma\, e \mid \mathsf{noop}\, e \\ v \quad &= i \mid \langle v, v \rangle \mid \lambda x.\, e \mid \lambda(x : \tau).\, e \mid \lambda(x : \mathsf{scan}\, \sigma).\, e \mid \\ &\quad\ \mathbb{G}\, (\tau \rightarrow \tau)\, v \\ \sigma \quad &= \mathsf{Nat} \mid \mathsf{Int} \mid \mathsf{Pair} \mid \mathsf{Fun} \mid \mathsf{Top} \\ \mathsf{Error} &= \mathsf{WrapErr} \mid \mathsf{ScanErr} \mid \mathsf{DivZeroErr} \mid \mathsf{TagErr} \\ E \quad &= \bullet \mid \mathit{unop}\, E \mid \mathit{binop}\, E\, e \mid \mathit{binop}\, v\, E \mid \langle E, e \rangle \mid \langle v, E \rangle \mid \\ &\quad\ \mathsf{app}\, E\, e \mid \mathsf{app}\, v\, E \mid \mathsf{noop}\, E \mid \mathsf{scan}\, \sigma\, E \mid \mathsf{wrap}\, \tau\, E \end{aligned}$$

Figure 5: Evaluation syntax

a shape annotation and a scan tag in the evaluation syntax ($\lambda(x : \mathsf{scan}\, \sigma).\, e$) to suggest that such functions must validate the shape of their input at run-time. Errors may arise from either a failed check at wrap boundary (WrapErr), a failed check at a scan boundary (ScanErr), a division by zero (DivZeroErr), or a malformed untyped expression (TagErr).

$\boxed{\Gamma \vdash_{\mathcal{D}} e : \tau}$

$$\frac{(x_0 : \tau_0) \in \Gamma}{\Gamma \vdash_{\mathcal{D}} x_0 : \tau_0} \qquad \frac{(x_0 : \tau_0), \Gamma \vdash_{\mathcal{D}} e_0 : \tau_1}{\Gamma \vdash_{\mathcal{D}} \lambda(x_0 : \tau_0).\, e_0 : \tau_0 \rightarrow \tau_1}$$

$$\frac{\Gamma \vdash_{\mathcal{U}} v_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{D}} \mathbb{G}\, \tau_0\, v_0 : \tau_0} \qquad \frac{\Gamma \vdash_{\mathcal{S}} v_0 : \sigma_0}{\Gamma \vdash_{\mathcal{D}} \mathbb{G}\, \tau_0\, v_0 : \tau_0}$$

$$\frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0 \rightarrow \tau_1 \quad \Gamma \vdash_{\mathcal{D}} e_1 : \tau_0}{\Gamma \vdash_{\mathcal{D}} \mathsf{app}\, e_0\, e_1 : \tau_1} \qquad \frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0}{\Gamma \vdash_{\mathcal{D}} \mathsf{noop}\, e_0 : \tau_0}$$

$$\frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{D}} \mathsf{wrap}\, \tau_0\, e_0 : \tau_0} \qquad \frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_0}{\Gamma \vdash_{\mathcal{D}} \mathsf{wrap}\, \tau_0\, e_0 : \tau_0}$$

Figure 6: Deep typing judgment (selected rules)

3.4 Three-Way Evaluation Typing

The evaluation syntax comes with three typing judgments that describe the invariants of deep, shallow, and untyped code. The deep typing judgment ($\vdash_{\mathcal{D}}$) validates full types, the shallow judgment ($\vdash_{\mathcal{S}}$) checks top-level type shapes, and the untyped judgment ($\vdash_{\mathcal{U}}$) checks that all variables are bound.

Both the deep and untyped rules are similar to the corresponding surface-language rules because they support equally-strong conclusions (full types and the uni-type). The shallow judgment is different because it validates type shapes instead of full types. When inspecting a pair, for example, the shallow judgment concludes with the Pair shape no matter what shapes the elements have. Consequently, a pair elimination form such as ($\mathsf{fst}\, x_0$) has the Top shape because the pair may contain any sort of value. Similar comments apply to functions and applications. Thus if a program expects a certain shape from a pair element or a function call, then the program must use a scan term to confirm the expectation.

3.5 Compilation from Surface to Evaluation

A compilation pass maps surface terms with modules to evaluation-language terms with run-time checks. The goal of the inserted checks is to ensure that well-typed surface expressions are well-typed in the evaluation syntax.

- In deep-typed code, all module boundaries to non-deep code become wrap checks. Compilation inserts no other checks.
- In shallow code, deep boundaries become wrap checks and untyped boundaries become scan checks. Extra scan checks protect typed code (e.g., from pairs).
- In untyped code, boundaries to deep modules become wrap checks and boundaries to shallow modules become scan checks. There are no other checks.

The rules in figure 9 present selected details of compilation. Variables compile to themselves. Functions in deep (and untyped) code simply recur on the function body and compile

$\boxed{\Gamma \vdash_{\mathcal{S}} e : \sigma}$

$$\frac{(x_0 : \sigma_0) \in \Gamma}{\Gamma \vdash_{\mathcal{S}} x_0 : \sigma_0} \qquad \frac{(x_0 : \mathcal{U}), \Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{S}} \lambda x_0.\, e_0 : \mathsf{Fun}}$$

$$\frac{(x_0 : \sigma_0), \Gamma \vdash_{\mathcal{S}} e_0 : \sigma_1}{\Gamma \vdash_{\mathcal{S}} \lambda(x_0 : \mathsf{scan}\, \sigma_0).\, e_0 : \mathsf{Fun}} \qquad \frac{\Gamma \vdash_{\mathcal{D}} v_0 : \tau_0}{\Gamma \vdash_{\mathcal{S}} \mathbb{G}\, \tau_0\, v_0 : \mathsf{Fun}}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_0 : \mathsf{Fun} \quad \Gamma \vdash_{\mathcal{S}} e_1 : \sigma_0}{\Gamma \vdash_{\mathcal{S}} \mathsf{app}\, e_0\, e_1 : \mathsf{Top}} \qquad \frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_0}{\Gamma \vdash_{\mathcal{S}} \mathsf{noop}\, e_0 : \sigma_0}$$

$$\frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{S}} \mathsf{noop}\, e_0 : \mathsf{Top}} \qquad \frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{S}} \mathsf{scan}\, \sigma_0\, e_0 : \sigma_0}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_1}{\Gamma \vdash_{\mathcal{S}} \mathsf{scan}\, \sigma_0\, e_0 : \sigma_0} \qquad \frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0 \quad \mathit{shape}(\tau_0) = \sigma_0}{\Gamma \vdash_{\mathcal{S}} \mathsf{wrap}\, \tau_0\, e_0 : \sigma_0}$$

$\boxed{\sigma <: \sigma}$ $$\frac{}{\mathsf{Nat} <: \mathsf{Int}} \qquad \frac{}{\sigma_0 <: \mathsf{Top}}$$

$\boxed{\mathit{shape} : \tau \longrightarrow \sigma}$

$$\begin{array}{llll} \mathit{shape}(\mathsf{Nat}) = \mathsf{Nat} & & \mathit{shape}(\tau_0 \times \tau_1) & = \mathsf{Pair} \\ \mathit{shape}(\mathsf{Int}) = \mathsf{Int} & & \mathit{shape}(\tau_0 \rightarrow \tau_1) & = \mathsf{Fun} \end{array}$$

Figure 7: Shallow typing (selected rules), subtyping, and type-to-shape metafunction

$\boxed{\Gamma \vdash_{\mathcal{U}} e : \mathcal{U}}$

$$\frac{(x_0 : \mathcal{U}) \in \Gamma}{\Gamma \vdash_{\mathcal{U}} x_0 : \mathcal{U}} \qquad \frac{(x_0 : \mathcal{U}), \Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{U}} \lambda x_0.\, e_0 : \mathcal{U}}$$

$$\frac{(x_0 : \sigma_0), \Gamma \vdash_{\mathcal{S}} e_0 : \sigma_1}{\Gamma \vdash_{\mathcal{U}} \lambda(x_0 : \mathsf{scan}\, \sigma_0).\, e_0 : \mathcal{U}} \qquad \frac{\Gamma \vdash_{\mathcal{D}} v_0 : \tau_0}{\Gamma \vdash_{\mathcal{U}} \mathbb{G}\, \tau_0\, v_0 : \mathcal{U}}$$

$$\frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U} \quad \Gamma \vdash_{\mathcal{U}} e_1 : \mathcal{U}}{\Gamma \vdash_{\mathcal{U}} \mathsf{app}\, e_0\, e_1 : \mathcal{U}} \qquad \frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0}{\Gamma \vdash_{\mathcal{U}} \mathsf{wrap}\, \tau_0\, e_0 : \mathcal{U}}$$

Figure 8: Untyped typing judgment (selected rules)

$\boxed{\Gamma \vdash_{\mathsf{s}} s : T \leadsto e}$

$$\frac{}{\Gamma \vdash_{\mathsf{s}} x_0 : T_0 \leadsto x_0} \qquad \frac{(x_0 : \tau_0), \Gamma \vdash_{\mathsf{s}} e_0 : \tau_1 \leadsto e_1}{\Gamma \vdash_{\mathsf{s}} \lambda(x_0 : \tau_0).\, e_0 : \tau_0 \rightarrow \tau_1 \leadsto \lambda(x_0 : \tau_0).\, e_1}$$

$$\frac{(x_0 : \lfloor \tau_0 \rfloor), \Gamma \vdash_{\mathsf{s}} e_0 : \lfloor \tau_1 \rfloor \leadsto e_1 \quad \mathit{shape}(\tau_0) = \sigma_0}{\Gamma \vdash_{\mathsf{s}} \lambda(x_0 : \lfloor \tau_0 \rfloor).\, e_0 : \lfloor \tau_0 \rightarrow \tau_1 \rfloor \leadsto \lambda(x_0 : \mathsf{scan}\, \sigma_0).\, e_1}$$

$$\frac{\Gamma \vdash_{\mathsf{s}} e_0 : \tau_1 \rightarrow \tau_0 \leadsto e_2 \quad \Gamma \vdash_{\mathsf{s}} e_1 : \tau_1 \leadsto e_3}{\Gamma \vdash_{\mathsf{s}} \mathsf{app}\, e_0\, e_1 : \tau_0 \leadsto \mathsf{app}\, e_2\, e_3}$$

$$\frac{\begin{array}{c} \Gamma \vdash_{\mathsf{s}} e_0 : \lfloor \tau_1 \rightarrow \tau_0 \rfloor \leadsto e_2 \\ \Gamma \vdash_{\mathsf{s}} e_1 : \lfloor \tau_1 \rfloor \leadsto e_3 \quad \mathit{shape}(\tau_0) = \sigma_0 \end{array}}{\Gamma \vdash_{\mathsf{s}} \mathsf{app}\, e_0\, e_1 : \lfloor \tau_0 \rfloor \leadsto \mathsf{scan}\, \sigma_0\, (\mathsf{app}\, e_2\, e_3)}$$

$$\left[\frac{\Gamma \vdash_{\mathsf{s}} e_0 : T_0 \leadsto e_1}{\Gamma \vdash_{\mathsf{s}} \mathsf{module}\, L_0\; e_0 : T_1 \leadsto e_2} \right]$$

L_0	T_0	T_1	$\leadsto e_2$
$\mathcal{D}$	τ_0	$\mathcal{U}$	$\mathsf{wrap}\, \tau_0\; e_1$
$\mathcal{S}$	$\lfloor \tau_0 \rfloor$	τ_0	$\mathsf{wrap}\, \tau_0\; e_1$
$\mathcal{U}$	$\mathcal{U}$	τ_0	$\mathsf{wrap}\, \tau_0\; e_1$
$\mathcal{D}$	τ_0	$\lfloor \tau_0 \rfloor$	$\mathsf{wrap}\, \tau_0\; e_1$
$\mathcal{S}$	$\lfloor \tau_0 \rfloor$	$\mathcal{U}$	$\mathsf{noop}\; e_1$
$\mathcal{U}$	$\mathcal{U}$	$\lfloor \tau_0 \rfloor$	$\mathsf{scan}\, \sigma_0\; e_1$

where $\sigma_0 = \mathit{shape}(\tau_0)$

Figure 9: Surface-to-evaluation compilation (selected rules)

to a new function. Functions in shallow code add a scan tag to their argument to indicate the need for a domain check, because untyped code can potentially invoke these functions. Applications in deep (and untyped) code recur on their subexpressions. Applications in shallow code insert an additional scan check to validate the result. Pair elimination forms (fst, snd) use scans in a similar way. Finally, one template rule and a table represent six rules for module boundaries. These rules correspond to arrows in figure 2.

Example. The three-module program from figure 1 can be encoded with shallow types roughly as follows:

$$\begin{array}{l} \quad \mathsf{let}\; x_0 = \mathsf{module}\, \mathcal{S}\; (\mathsf{module}\, \mathcal{U}\; (\lambda x_0\, x_1.\, \_))\; \mathsf{in} \\ \qquad \mathsf{app}\, x_0\; \texttt{`cat'}\; \texttt{`roman'} \end{array}$$

Compilation yields a term with one scan check:

$$\begin{array}{l} \quad \mathsf{let}\; x_0 = \mathsf{noop}\; (\mathsf{scan}\, \mathsf{Fun}\; (\lambda x_0\, x_1.\, \_))\; \mathsf{in} \\ \qquad \mathsf{app}\, x_0\; \texttt{`cat'}\; \texttt{`roman'} \end{array}$$

3.6 Reduction Relation

The left half of figure 10 presents a notion of reduction for the evaluation syntax. (Section 3.7 discusses the right half.) Each rule relates two expressions ($e \rhd e$). Rules that share a syntactically-equal domain come with a test for the domain expression. These tests use basic set theory to pattern-match on expressions; for example, the test ($v_0 \in \lambda(x : \tau).\, e$) holds when the value v_0 is a type-annotated lambda.

The rules for unary and binary operations apply the δ metafunction (figure 11) and halt with a tag error if δ is undefined. In general, δ models the behavior of a run-time system that works at a lower level of abstraction than the evaluation language. For unary operations, δ eliminates a pair. For binary operations, δ performs arithmetic.

The rules for function application check that the first expression is a function and try to substitute the argument expression into the function body. If the function has a type-shape annotation (σ), then a shape check (figure 11) validates the argument before substitution. If the function is enclosed in a guard wrapper, then the application unfolds into two wrap checks: one for the argument and one for the

$\boxed{e \triangleright e}$

$\mathit{unop}\, v_0 \quad \triangleright \mathsf{TagErr}$
if $\delta(\mathit{unop}, v_0)$ is undefined

$\mathit{unop}\, v_0 \quad \triangleright \delta(\mathit{unop}, v_0)$
if $\delta(\mathit{unop}, v_0)$ is defined

$\mathit{binop}\, v_0\, v_1 \quad \triangleright \mathsf{TagErr}$
if $\delta(\mathit{binop}, v_0, v_1)$ is undefined

$\mathit{binop}\, v_0\, v_1 \quad \triangleright \delta(\mathit{binop}, v_0, v_1)$
if $\delta(\mathit{binop}, v_0, v_1)$ is defined

$\mathsf{app}\, v_0\, v_1 \quad \triangleright \mathsf{TagErr}$
if $\neg \mathit{shape\text{-}match}(\mathsf{Fun}, v_0)$

$\mathsf{app}\, (\lambda x_0.\, e_0)\, v_0 \quad \triangleright e_0[x_0 \leftarrow v_0]$

$\mathsf{app}\, (\lambda (x_0 : \tau_0).\, e_0)\, v_0 \quad \triangleright e_0[x_0 \leftarrow v_0]$

$\mathsf{app}\, (\lambda (x_0 : \mathsf{scan}\, \sigma_0).\, e_0)\, v_0 \triangleright \mathsf{ScanErr}$
if $\neg \mathit{shape\text{-}match}(\sigma_0, v_0)$

$\mathsf{app}\, (\lambda (x_0 : \mathsf{scan}\, \sigma_0).\, e_0)\, v_0 \triangleright e_0[x_0 \leftarrow v_0]$
if $\mathit{shape\text{-}match}(\sigma_0, v_0)$

$\mathsf{app}\, (\mathbb{G}\, (\tau_0 \longrightarrow \tau_1)\, v_0)\, v_1 \quad \triangleright$
$\mathsf{wrap}\, \tau_1\, (\mathsf{app}\, v_0\, (\mathsf{wrap}\, \tau_0\, v_1))$

$\mathsf{noop}\, v_0 \quad \triangleright v_0$

$\mathsf{scan}\, \sigma_0\, v_0 \quad \triangleright \mathsf{ScanErr}$
if $\neg \mathit{shape\text{-}match}(\sigma_0, v_0)$

$\mathsf{scan}\, \sigma_0\, v_0 \quad \triangleright v_0$
if $\mathit{shape\text{-}match}(\sigma_0, v_0)$

$\mathsf{wrap}\, \tau_0\, v_0 \quad \triangleright \mathsf{WrapErr}$
if $\neg \mathit{shape\text{-}match}(\mathit{shape}(\tau_0), v_0)$

$\mathsf{wrap}\, (\tau_0 \longrightarrow \tau_1)\, v_0 \quad \triangleright \mathbb{G}\, (\tau_0 \longrightarrow \tau_1)\, v_0$
if $\mathit{shape\text{-}match}(\mathsf{Fun}, v_0)$

$\mathsf{wrap}\, (\tau_0 \times \tau_1)\, \langle v_0, v_1 \rangle \quad \triangleright \langle \mathsf{wrap}\, \tau_0\, v_0, \mathsf{wrap}\, \tau_1\, v_1 \rangle$

$\mathsf{wrap}\, \tau_0\, v_0 \quad \triangleright v_0$
if $\tau_0 \in \mathsf{Int} \cup \mathsf{Nat}$ and $\mathit{shape\text{-}match}(\tau_0, v_0)$

$\boxed{e \longrightarrow^{*} e} \overset{def}{=}$ reflexive, transitive, compatible (w.r.t. E) closure of $\triangleright$

$\boxed{s \longrightarrow^{*} e} \overset{def}{=} \exists\, T, e_1.$
$\vdash_s s : T \rightsquigarrow e_1 \wedge e_1 \longrightarrow^{*} e$

$\boxed{(e)^{\ell} \triangleright^{+} (e)^{\ell}}$

$(\mathit{unop}\, \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0})^{\ell_1} \quad \triangleright^{+} (\mathsf{TagErr})^{\ell_1}$
if $v_0 \notin (v)^{\ell}$ and $\delta(\mathit{unop}, v_0)$ is undefined

$(\mathit{unop}\, \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0})^{\ell_1} \quad \triangleright^{+} \llparenthesis \delta(\mathit{unop}, v_0) \rrparenthesis^{\overline{\ell}_0 \ell_1}$
if $\delta(\mathit{unop}, v_0)$ is defined

$(\mathit{binop}\, \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0}\, \llparenthesis v_1 \rrparenthesis^{\overline{\ell}_1})^{\ell_2} \quad \triangleright^{+} (\mathsf{TagErr})^{\ell_2}$
if $v_i \notin (v)^{\ell}$ and $\delta(\mathit{binop}, v_0, v_1)$ is undefined

$(\mathit{binop}\, \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0}\, \llparenthesis v_1 \rrparenthesis^{\overline{\ell}_1})^{\ell_2} \quad \triangleright^{+} (\delta(\mathit{binop}, v_0, v_1))^{\ell_2}$
if $\delta(\mathit{binop}, v_0, v_1)$ is defined

$(\mathsf{app}\, \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0}\, v_1)^{\ell_1} \quad \triangleright^{+} (\mathsf{TagErr})^{\ell_1}$
if $v_0 \notin (v)^{\ell}$ and $\neg \mathit{shape\text{-}match}(\mathsf{Fun}, v_0)$

$(\mathsf{app}\, \llparenthesis \lambda x_0.\, e_0 \rrparenthesis^{\overline{\ell}_0}\, v_0)^{\ell_1} \quad \triangleright^{+} \llparenthesis e_0[x_0 \leftarrow \llparenthesis v_0 \rrparenthesis^{\ell_1 \mathit{rev}(\overline{\ell}_0)}] \rrparenthesis^{\overline{\ell}_0 \ell_1}$

$(\mathsf{app}\, \llparenthesis \lambda (x_0 : \tau_0).\, e_0 \rrparenthesis^{\overline{\ell}_0}\, v_0)^{\ell_1} \quad \triangleright^{+} \llparenthesis e_0[x_0 \leftarrow \llparenthesis v_0 \rrparenthesis^{\ell_1 \mathit{rev}(\overline{\ell}_0)}] \rrparenthesis^{\overline{\ell}_0 \ell_1}$

$(\mathsf{app}\, \llparenthesis \lambda (x_0 : \mathsf{scan}\, \sigma_0).\, e_0 \rrparenthesis^{\overline{\ell}_0}\, v_0)^{\ell_1} \quad \triangleright^{+} (\mathsf{ScanErr})^{\ell_1}$
if $\neg \mathit{shape\text{-}match}(\sigma_0, v_0)$

$(\mathsf{app}\, \llparenthesis \lambda (x_0 : \mathsf{scan}\, \sigma_0).\, e_0 \rrparenthesis^{\overline{\ell}_0}\, v_0)^{\ell_1} \quad \triangleright^{+} \llparenthesis e_0[x_0 \leftarrow \llparenthesis v_0 \rrparenthesis^{\ell_1 \mathit{rev}(\overline{\ell}_0)}] \rrparenthesis^{\overline{\ell}_0 \ell_1}$
if $\mathit{shape\text{-}match}(\sigma_0, v_0)$

$(\mathsf{app}\, \llparenthesis \mathbb{G}\, (\tau_0 \longrightarrow \tau_1)\, (v_0)^{\ell_0} \rrparenthesis^{\overline{\ell}_1}\, v_1)^{\ell_2} \quad \triangleright^{+}$
$\llparenthesis (\mathsf{wrap}\, \tau_1\, (\mathsf{app}\, v_0\, (\mathsf{wrap}\, \tau_0\, \llparenthesis v_1 \rrparenthesis^{\ell_2 \mathit{rev}(\overline{\ell}_1)})))^{\ell_0} \rrparenthesis^{\overline{\ell}_1 \ell_2}$

$(\mathsf{noop}\, \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0})^{\ell_1} \quad \triangleright^{+} \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0 \ell_1}$

$(\mathsf{scan}\, \sigma_0\, \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0})^{\ell_1} \quad \triangleright^{+} (\mathsf{ScanErr})^{\ell_1}$
if $\neg \mathit{shape\text{-}match}(\sigma_0, v_0)$

$(\mathsf{scan}\, \sigma_0\, \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0})^{\ell_1} \quad \triangleright^{+} \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0 \ell_1}$
if $\mathit{shape\text{-}match}(\sigma_0, v_0)$

$(\mathsf{wrap}\, \tau_0\, \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0})^{\ell_1} \quad \triangleright^{+} (\mathsf{WrapErr})^{\ell_1}$
if $\neg \mathit{shape\text{-}match}(\mathit{shape}(\tau_0), v_0)$

$(\mathsf{wrap}\, (\tau_0 \longrightarrow \tau_1)\, \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0})^{\ell_1} \quad \triangleright^{+} (\mathbb{G}\, (\tau_0 \longrightarrow \tau_1)\, \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0})^{\ell_1}$
if $\mathit{shape\text{-}match}(\mathsf{Fun}, v_0)$

$(\mathsf{wrap}\, (\tau_0 \times \tau_1)\, \llparenthesis \langle v_0, v_1 \rangle \rrparenthesis^{\overline{\ell}_0})^{\ell_1} \quad \triangleright^{+} (\langle \mathsf{wrap}\, \tau_0\, \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0}, \mathsf{wrap}\, \tau_1\, \llparenthesis v_1 \rrparenthesis^{\overline{\ell}_0} \rangle)^{\ell_1}$

$(\mathsf{wrap}\, \tau_0\, \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0})^{\ell_1} \quad \triangleright^{+} (v_0)^{\ell_1}$
if $\tau_0 \in \mathsf{Int} \cup \mathsf{Nat}$ and $\mathit{shape\text{-}match}(\tau_0, v_0)$

$\boxed{e \overset{+}{\longrightarrow}{}^{*}\, e} \overset{def}{=}$ reflexive, transitive, compatible (w.r.t. E) closure of $\triangleright^{+}$

Figure 10: Semantics for the evaluation syntax (left) and a labeled variant (right)

$$\boxed{\delta : unop \times v \longrightarrow v}$$

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

$$\boxed{\delta : binop \times v \times v \longrightarrow v}$$

$$\begin{aligned} \delta(\mathsf{plus}, i_0, i_1) &= i_0 + i_1 \\ \delta(\mathsf{quotient}, i_0, 0) &= \mathsf{DivErr} \\ \delta(\mathsf{quotient}, i_0, i_1) &= \lfloor i_0 / i_1 \rfloor \end{aligned}$$

$$\boxed{shape\text{-}match : \sigma \times v \longrightarrow \mathcal{B}}$$

$$\begin{aligned} & shape\text{-}match(\mathsf{Fun}, v_0) && = \mathsf{True} \\ & \quad \text{if } v_0 \in \lambda x.\, e \cup \lambda(x\!:\!\tau).\, e \cup \lambda(x\!:\!\mathsf{scan}\,\sigma).\, e \cup \mathbb{G}\,\tau\,v \\ & shape\text{-}match(\mathsf{Pair}, \langle v_0, v_1\rangle) && = \mathsf{True} \\ & shape\text{-}match(\mathsf{Int}, i_0) && = \mathsf{True} \\ & shape\text{-}match(\mathsf{Nat}, n_0) && = \mathsf{True} \\ & shape\text{-}match(\mathsf{Top}, v_0) && = \mathsf{True} \\ & shape\text{-}match(\sigma_0, v_0) && = \mathsf{False} \\ & \quad \text{otherwise} \end{aligned}$$

$$\boxed{rev : \overline{\ell} \longrightarrow \overline{\ell}} \quad rev(\ell_0, \cdots, \ell_n) = \ell_n, \cdots, \ell_0$$

Figure 11: Semantic metafunctions

result. Functions that are wrapped in several guards must step through several unfoldings.

The remaining rules state the behavior of run-time checks. A noop boundary performs no check and lets any value across. A scan boundary checks the top-level shape of an incoming value against the expected type-shape, and halts if the two disagree. Lastly, a wrap boundary checks the top-level shape of a value and then proceeds based on the type. For function types, a wrap installs a guard wrapper. For pairs, a wrap validates both components and creates a new pair value. For base types, the shape check is enough.

The semantics of the evaluation syntax is given in standard fashion [14] as the the reflexive, transitive closure of the compatible closure of $\rhd$ relative to the evaluation contexts (E) from figure 5. Each expression has a unique redex thanks to the inductive structure of evaluation contexts.

3.7 Labeled Evaluation, Deep Label Consistency

The model requires two final definitions to enable a syntactic analysis of complete monitoring: a label-annotated reduction relation and a consistency judgment that validates the labels. Labels provide a specification of who owns what in a running program. More precisely, the labels on an expression describe the surface modules that are responsible for the behavior of the expression. A consistently-labeled expression keeps deep-typed code separate from shallow and untyped code. Informally, consistent labelling is possible if a semantics can check all inputs to and outputs from deep-typed values.

$$\begin{aligned} e &= x \mid v \mid \langle e, e\rangle \mid unop\, e \mid binop\, e\, e \mid \mathsf{app}\, e\, e \mid \mathsf{Error} \mid \\ &\quad\; \mathsf{wrap}\,\tau\,(e)^{\ell} \mid \mathsf{scan}\,\sigma\,(e)^{\ell} \mid \mathsf{noop}\,(e)^{\ell} \mid (e)^{\ell} \\ v &= i \mid \langle v, v\rangle \mid \lambda x.\, e \mid \lambda(x\!:\!\tau).\, e \mid \lambda(x\!:\!\mathsf{scan}\,\sigma).\, e \mid \\ &\quad\; \mathbb{G}\,(\tau \rightarrow \tau)\,(v)^{\ell} \mid (v)^{\ell} \\ E &= \ldots \mid (E)^{\ell} \\ \ell &= \mathcal{D}_0 \mid \mathcal{D}_1 \mid \ldots \mid \mathcal{S}_0 \mid \mathcal{S}_1 \mid \ldots \mid \mathcal{U}_0 \mid \mathcal{U}_1 \mid \ldots \\ \overline{\ell} &= \text{sequence of labels } (\ell) \\ \mathcal{L} &= \cdot \mid (x\!:\!\ell), \mathcal{L} \end{aligned}$$

$$\text{Abbreviation: } (\cdots(e_0)^{\ell_0}\cdots)^{\ell_n} = \llparenthesis e_0 \rrparenthesis^{\ell_0, \cdots, \ell_n}$$

Figure 12: Labeled evaluation syntax

The right half of figure 10 presents a labeled notion of reduction for the evaluation language.[3] By design, the reduction rules are identical to the basic rules from figure 10 except for superscript labels and parentheses. Labels are metadata; they do not change the underlying behavior of a reduction rule. The labels on the left-hand expression of each rule give names to the parties responsible for any relevant subexpressions. The labels on the right-hand expression show how responsibilities change in response to the reduction step. For example, an untyped function application ($\mathsf{app}\,(\lambda x_0.\, e_0)\, v_0$) substitutes an argument value into the function body. Because of the substitution, the parties that were responsible for the function become responsible for both the value and for the expression that the function computes. The label metafunction *rev* (figure 11) keeps these labels in proper order by reversing them—because the argument value flows in to the function.

Labels typically accumulate without bound. The only way that labels may disappear is after a successful run-time check or after an error (when evaluation is over). For example, the wrap rule for base types says that client ℓ_1 may assume full responsibility of numbers that reach a well-typed boundary.

Technically, the addition of labels to the evaluation language calls for an entirely new syntax (figure 12). The expression form $(e)^{\ell}$ attaches a label to any subexpression. A similar value form $(v)^{\ell}$ lets any value appear under an arbitrary number of labels. These labels correspond to modules from the surface syntax, and thus combine a kind ($\mathcal{D}$, $\mathcal{S}$, or $\mathcal{U}$) with a unique identifying number. The labeled syntax has two other noteworthy aspects:

- All boundaries require a label for their subexpression. This means that the v_0 in the following four patterns must have at least one label: ($\mathsf{wrap}\;\tau_0\; v_0$), ($\mathsf{scan}\;\sigma_0\; v_0$), ($\mathsf{noop}\; v_0$), and ($\mathbb{G}\;\tau_0\; v_0$).
- To reduce parenthesis and superscripts, the abbrevation $\llparenthesis \cdot \rrparenthesis^{\cdot}$ captures a sequence of labels. For example,

[3] The design of a labeled reduction relation is like any other definition in that it requires ingenuity to create and careful reading to understand. To help readers gain an intuition for appropriate labeling, the appendix presents the guidelines that underlie figure 10.

$$\boxed{\ell; \mathcal{L} \Vdash e}$$

$$\frac{\ell_1; \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \mathsf{noop}\ (e_0)^{\ell_1}} \qquad \frac{\ell_1; \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \mathsf{scan}\, \sigma_0\ (e_0)^{\ell_1}} \qquad \frac{\ell_1; \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \mathsf{wrap}\, \tau_0\ (e_0)^{\ell_1}} \qquad \frac{\ell_1; \mathcal{L}_0 \Vdash v_0}{\ell_0; \mathcal{L}_0 \Vdash \mathbb{G}\, \tau_0\ (v_0)^{\ell_1}} \qquad \frac{\mathcal{D}_1; \mathcal{L}_0 \Vdash e_0}{\mathcal{D}_0; \mathcal{L}_0 \Vdash (e_0)^{\mathcal{D}_1}}$$

$$\frac{\mathcal{S}_1; \mathcal{L}_0 \Vdash e_0}{\mathcal{S}_0; \mathcal{L}_0 \Vdash (e_0)^{\mathcal{S}_1}} \qquad \frac{\mathcal{U}_0; \mathcal{L}_0 \Vdash e_0}{\mathcal{S}_0; \mathcal{L}_0 \Vdash (e_0)^{\mathcal{U}_0}} \qquad \frac{\mathcal{U}_1; \mathcal{L}_0 \Vdash e_0}{\mathcal{U}_0; \mathcal{L}_0 \Vdash (e_0)^{\mathcal{U}_1}} \qquad \frac{\mathcal{S}_0; \mathcal{L}_0 \Vdash e_0}{\mathcal{U}_0; \mathcal{L}_0 \Vdash (e_0)^{\mathcal{S}_0}}$$

Figure 13: Deep label consistency (selected rules)

the value $(((4)^{\ell_0})^{\ell_1})^{\ell_2}$ matches the pattern $\llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0}$ with $v_0 = 4$ and $\overline{\ell}_0 = \ell_0, \ell_1, \ell_2$.

Figure 13 presents a consistency judgment for labeled expressions. The judgment allows any mix of shallow ($\mathcal{S}$) and untyped ($\mathcal{U}$) labels around an expression, but restricts the use of deep labels ($\mathcal{D}$). Concretely, the judgment analyzes an expression relative to a context label and an environment ($\mathcal{L}$). Variables must have a binding in the label environment that matches the context label and most other expressions simply need consistent subterms; these rules are deferred to the appendix. Boundary expressions and guarded values are ownership switch points; these terms are consistent if their subterm matches the context label that appears inside the boundary. The rules for labeled expressions specify the allowed mixtures. Shallow and untyped labels can mix together around an expression, but a deep-labeled expression must have only deep labels around it.

3.8 Properties

Type soundness predicts the possible outcomes of a well-typed expression. Because the surface language allows three kinds of typed expression (deep, shallow, and untyped), the definition is parameterized over both a language kind L and a characterization function F that maps a *subset* of the surface types T to an evaluation-language type (either τ, σ, or $\mathcal{U}$).

Definition 3.1 (**TS** ($\vdash_L$, F))**.** *Language L satisfies* **TS** ($\vdash_L$, F) *if for all s_0 such that $\vdash_s s_0 : T$ holds and $\mathsf{F}(T)$ is defined, one of the following holds:*

- $s_0 \longrightarrow^* v_0$ *and* $\vdash_L v_0 : \mathsf{F}(T)$
- $s_0 \longrightarrow^*$ Error
- $s_0 \longrightarrow^*$ *diverges*

There are three important characterization functions F for the analysis: **1** is the identity function on types τ; **shape** maps underlined types $\underline{\tau}$ to shapes σ (similar to *shape* from figure 7); and **0** maps $\mathcal{U}$ to $\mathcal{U}$.

Theorem 3.2 (type soundness)**.**

- *Language $\mathcal{D}$ satisfies* **TS** ($\vdash_{\mathcal{D}}$, **1**)
- *Language $\mathcal{S}$ satisfies* **TS** ($\vdash_{\mathcal{S}}$, **shape**)
- *Language $\mathcal{U}$ satisfies* **TS** ($\vdash_{\mathcal{U}}$, **0**)

Proof Sketch. By lemmas for progress, preservation, and compilation (deferred to the appendix). □

Unlike a conventional soundness theorem [41, 71], definition 3.1 does not claim that the evaluation of a well-typed expression cannot go wrong by throwing a tag error. Such a claim would not hold for typed expressions that contain faulty untyped modules. It is true, however, that the reduction of a well-typed redex cannot yield a tag error:

Lemma 3.3 (type discipline)**.** *If e_0 is typed (either $\vdash_{\mathcal{D}} e_0 : \tau_0$ or $\vdash_{\mathcal{S}} e_0 : \sigma_0$) and $e_0 \rhd e_1$ then $e_1 \notin$* TagErr.

Complete monitoring states that the evaluation language has control over every interaction between deep-typed code and weaker code. More precisely, the proof-technical question is whether the labels that arise in evaluation are consistent according to the $\Vdash$ judgment (figure 13).

Theorem 3.4 (complete monitoring)**.** *If $\vdash_s s_0 : T \rightsquigarrow e_0$ and $\ell_0; \cdot \Vdash e_0$ and $e_0 \xrightarrow{+}{}^* e_1$ then $\ell_0; \cdot \Vdash e_1$.*

Proof Sketch. By a preservation argument. The proofs for a few interesting cases are sketched below. Other cases are in the appendix.

Case: $(\mathsf{app}\ \llparenthesis \lambda x_0.\, e_0 \rrparenthesis^{\overline{\ell}_0}\ v_0)^{\ell_1} \rhd^{+} \llparenthesis e_0[x_0 \leftarrow \llparenthesis v_0 \rrparenthesis^{\ell_1 rev(\overline{\ell}_0)}] \rrparenthesis^{\overline{\ell}_0 \ell_1}$

1. $\overline{\ell}_0$ is all deep or a mix of shallow and untyped, by deep-label consistency of the redex
2. $\ell_1; \cdot \Vdash v_0$, also by deep-label consistency of the redex
3. let ℓ_n be the rightmost label in the sequence $\overline{\ell}_0$
4. $\ell_n; \cdot \Vdash \llparenthesis v_0 \rrparenthesis^{\ell_1 rev(\overline{\ell}_0)}$, by steps 1 and 2
5. $\ell_n; \cdot \Vdash x_0$ for each occurrence of x_0 in e_0, by deep-label consistency of the redex
6. by a substitution lemma

Case: $(\mathsf{app}\ \llparenthesis \mathbb{G}\ \tau_0 \rightarrow \tau_1\ (v_0)^{\ell_0} \rrparenthesis^{\overline{\ell}_1}\ v_1)^{\ell_2} \rhd^{+}$
$\llparenthesis (\mathsf{wrap}\, \tau_1\ (\mathsf{app}\, v_0\ (\mathsf{wrap}\, \tau_0\ \llparenthesis v_1 \rrparenthesis^{\ell_2 rev(\overline{\ell}_1)})))^{\ell_0} \rrparenthesis^{\overline{\ell}_1 \ell_2}$

1. $\ell_0; \cdot \Vdash v_0$, by deep-label consistency of the redex
2. $\ell_2; \cdot \Vdash v_1$, again by deep-label consistency
3. $\overline{\ell}_1$ is either all deep or a mix of shallow and untyped, again by the consistency of the redex
4. by the definition of $\Vdash$

Case: $(\mathsf{noop}\ \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0})^{\ell_1} \rhd^{+} \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0 \ell_1}$
by the definition of $\rightsquigarrow$, because a noop boundary connects either two deep components or a mix of shallow and untyped components (self edges or $\mathcal{S}$ to $\mathcal{U}$)

Case: $(\mathsf{scan}\, \sigma_0\ \llparenthesis v_0 \rrparenthesis^{\overline{\ell}_0})^{\ell_1} \rhd^{+} (\mathsf{ScanErr})^{\ell_1}$

by the definition of $\Vdash$

Case: $(\mathsf{scan}\,\sigma_0\ (\!(v_0)\!)^{\overline{\ell}_0})^{\ell_1} \rhd^{+} (\!(v_0)\!)^{\overline{\ell}_0 \ell_1}$
by the definition of $\rightsquigarrow$, because a scan boundary links only shallow and/or untyped components

Case: $(\mathsf{wrap}\,(\tau_0 \rightarrow \tau_1)\ (\!(v_0)\!)^{\overline{\ell}_0})^{\ell_1} \rhd^{+} (\mathbb{G}\ \tau_0 \rightarrow \tau_1\ (\!(v_0)\!)^{\overline{\ell}_0})^{\ell_1}$
by the definition of $\Vdash$

□

4 Implementation Challenges

We have implemented three-way interactions atop Typed Racket. The extension combines the standard "Deep" Typed Racket, which implements the natural semantics [62], with the "Shallow Racket" implementation of transient [28]. Programmers may choose deep or shallow types when declaring a module. Switching between the two is a one-line change except in programs that fine-tune the checks that guard type boundaries (section 4.4).

For the most part, the model was an effective guide for the implementation. Deep and Shallow share a common surface syntax, type checker, and evaluation syntax. The key issue was how to modify these compiler back-ends to produce code with context-dependent runtime checks. Unexpected challenges arose regarding separate compilation, the enforcement of deep types, and metaprogramming.

4.1 Wrapping Contracts and Type Environments

Higher-order exports from deep-typed code need protection from untyped and shallow-typed clients. Wrapping contracts are a convenient way to implement this protection because they let deep modules share exports with no performance overhead. They introduce a problem with separate compilation, however, because the type checker for shallow code must find a type for these wrappers to understand uses of deep-typed identifiers.

In Typed Racket, all exports from deep code statically resolve to either an unwrapped identifier or a wrapped one depending on the context in which they are used [10, 59]. The wrappers do not have types due to the organization of compiled code. Types appear in one submodule [19] while wrappers appear in a sibling submodule to delay the cost of building them. But because the wrappers are implemented as Racket contracts [17], they come with a compile-time pointer to the unwrapped identifier. Shallow Racket follows these pointers to typecheck interactions.

4.2 Shallow-to-Deep Contracts

Deep-typed code needs to wrap imports from untyped and shallow-typed modules. Because untyped imports lack types, the straightforward solution is to ask programmers for a type-annotated import statement and to generate contracts at the import. Shallow imports already have types. This raises a question about where to prepare the validating contracts: the exporting shallow module or the importing deep module.

Shallow Racket eagerly prepares contracts for its deep-typed clients and stores these contracts in a lazily-loaded submodule. The main benefit of this approach is that multiple clients can reference one set of contract definitions.

4.3 Macros and Hidden Exports

Macro expansion may cause private identifiers from one module to appear in the expansion of another module [18, 20]. If one module uses deep types and the other uses shallow, this behavior is a threat to type soundness. The stowed identifiers must be protected like any other export.

By default, Deep and Shallow Racket cannot share macros. Programmers can enable reuse by exporting a macro unsafely. An open question is whether a static analysis can determine which macros may safely cross type boundaries.

4.4 Three-Way Boundary Utilities

Static types and higher-order contracts are fundamentally different tools. Types enable proofs via static analysis. Contracts check behaviors dynamically. For certain types, such as a type for terminating functions [46], it is difficult to generate an approximating contract. A language may therefore wish to offer an API that lets programmers specify the contracts that enforce deep types at a boundary. These APIs must be adapted to support a three-way implementation.

Typed Racket comes with two tools for type boundaries. The first, `require/untyped-contract`, expects a typed identifier and a subtype of the identifier's actual type; it uses the subtype to generate a contract. This behavior can make it somewhat harder to switch from Deep to Shallow types. For example, the standard array library uses this tool to give untyped code access to an overloaded function that expects either an array of integers or an array of natural numbers. Rather than generate a contract based on the overloaded type, which would require a higher-order union contract, the library uses a subtype that expects arrays of integers. Shallow code can access this array function as well, but only through the contract. Switching a module from Deep to Shallow may therefore require casts to meet the subtype.

The second tool combines two identifiers. In the following example, `f` is defined as a context-sensitive identifier that expands to `tf` in Deep code and to `uf` in untyped code: `(define-typed/untyped-identifier f tf uf)`. Shallow cannot be trusted with `tf` because of its weak soundness guarantee, and it cannot use `uf` if that identifier lacks a type. Thus, the tool needs a third input for Shallow contexts.

5 Evaluation

The integration of Deep and Shallow Typed Racket offers substantial benefits over either one alone:

- Switching from Shallow to Deep strengthens the formal guarantees for a block of code (section 5.1).

- Switching from Deep to Shallow can remove spurious errors from a program (section 5.2).
- The combination of Deep and Shallow improves worst-case overheads relative to untyped code (section 5.3).

5.1 Guarantees by Deep

By design, deep types enforce stronger guarantees than shallow. A deep type is a behavioral claim that is substantiated by comprehensive run-time checks. No matter where a deep-typed value ends up, the type constrains its behavior. A shallow type is valid only in a limited scope. If a value escapes to untyped or less-precisely typed code, e.g., via subtyping, then its original type gets forgotten (section 2.2).

Prior work suggests that the relative weakness of shallow types can lead to confusing situations. Lazarek et al. [39] performed an automated study of debugging in Typed Racket and found that Shallow blame errors are more likely to reach a dead end than Deep blame errors. Tunnell Wilson et al. [65] conducted a survey using a hypothetical gradual language and reported that participants found the behaviors allowed by shallow types "unexpected" more often than deep types. Migrating from shallow to deep types may therefore be an effective way of finding the root issue in a buggy program.

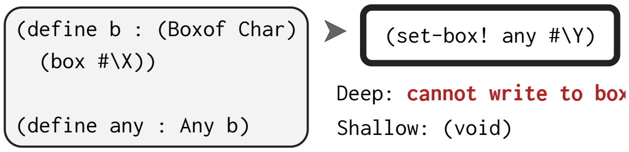
5.2 Expressiveness by Shallow

Shallow Racket can express a variety of useful programs that Deep Racket rejects at run-time. At first glance, the existence of such programs is surprising because the theory suggests that the deep semantics is more "correct" (section 3). It turns out that deep types can be overly restrictive; in such programs, delayed shallow checks work better in practice. In other programs, the gap between Deep and Shallow Racket is due to implementation issues. Refer to the appendix for motivating examples submitted by Typed Racket users.

5.2.1 Relaxed Top Type. Statically, the top type is a supertype of every other type. Programmers often use this type as a convenient placeholder to avoid committing to a more-specific type. When enforced as a deep type, however, the top type has a strict semantics that prevents clients from inspecting top-wrapped values [16]. For example, if deep code exports a function using a top type, then non-deep clients cannot invoke the function.

The shallow top type imposes no such restrictions. Untyped code may invoke a shallow function exported via the top type, and may even write to a top-typed array. These behaviors can be useful and do not undermine the weak shallow soundness guarantee.

Figure 14 presents an example in Typed Racket that uses a mutable box and the top type Any, which is *not* a dynamic type. When module in the left part of the figure uses Deep types, the untyped client cannot mutate the box. With Shallow, untyped mutations are allowed.



```
(define b : (Boxof Char)
  (box #\X))

(define any : Any b)
```

```
(set-box! any #\Y)
```

```
Deep: cannot write to box
Shallow: (void)
```

Figure 14: Deep Racket enforces the top type (Any) with a contract that rejects all inputs

5.2.2 No Missing Wrappers. Mutable values that can appear in deep code need tailored wrappers to monitor their interactions with non-deep clients. These wrappers are difficult to implement because they often require support from the run-time system [56]. Unsurprisingly, some infrequently-used types in Deep Racket lack wrappers (12 in total).

By contrast, a shallow language avoids the question of how to implement wrappers. Shallow types need only first-order checks, which require far less engineering.

5.2.3 Uniform Behavior. Although the purpose of deep wrappers is to reject type-incorrect operations without otherwise changing behaviors, certain wrappers in Deep Racket do cause subtle changes. The most problematic ones are the wrappers for polymorphic types. Deep Racket enforces types such as (All (A) (-> A A)) with a function contract that seals inputs and unseals outputs [33]. The seals change the outcome of basic operations.

Shallow Racket avoids all such changes in behavior, including the well-known object identity issues [56, 37, 68, 67], because the transient semantics does not use wrappers.

5.3 Performance by Deep and Shallow

The three-way mix of deep and shallow types improves performance across the board. On the GTP benchmark suite v6.0 [32], toggling between deep and shallow avoids pathological cases. Mixing deep and shallow modules can further improve performance, up to 2x faster than deep or shallow alone (relative to untyped code).

All data in this section was collected on a single-user Linux box with 4 physical i7-4790 3.60GHz cores and 16GB RAM. The machine ran Racket v7.8.0.5+ [48] and a pre-release of Typed Racket [66] that extends Typed Racket v1.12. Each data point is the result of running one program configuration nine times in a row and averaging the speed of the final eight runs. Our Racket [48] does not optimize transient checks to the same extent as a tracing JIT compiler (section 6), so there is potential room for improvement.

5.3.1 Deep and Shallow Combined. Mixing deep and shallow types within one program configuration can improve its performance. Such configurations are quite common in the GTP benchmarks. Out of the 2^N configurations in sixteen of the smaller benchmarks, a median of 37.5% run fastest with a mix of deep and shallow types (figure 15). These mixtures also increase the number of *D-deliverable migration paths*

Benchmark	Best w/ D+S
forth	12%
fsm	38%
fsmoo	31%
mbta	19%
morsecode	25%
zombie	6%
dungeon	31%
jpeg	38%
zordoz	47%
lnm	66%
suffixtree	48%
kcfa	55%
snake	46%
take5	36%
acquire	64%
tetris	62%

Figure 15: Percent of configurations that run fastest with a mix of Deep and Shallow modules.

(defined in section 5.3.4). All paths in fsm, morsecode, lnm, and kcfa become 1.2-deliverable when configurations can mix deep and shallow types.

These encouraging numbers are the result, however, of a search through 3^N configurations. The following three subsections therefore investigate Deep and Shallow mixtures without relying on an exhaustive search.

5.3.2 Case Studies. To test whether fast-running configurations can be found without a search, we manually explored deep and shallow combinations in the following programs:

MsgPack. MsgPack is a Typed Racket library that converts Racket values into serialized MessagePack data.[4] The author of this library reported poor performance due to deep type boundaries. Changing a bridge module from deep to shallow types (a one-line change), reduces the time needed to run all tests from 320 seconds to 204 seconds (40% speedup).

Synth. The synth benchmark is based on an untyped program that interacts with a deep-typed math library to synthesize music.[5] This untyped program runs 14x slower than a deep-typed version because of the library boundary. When the library uses shallow types instead, the gap between an untyped and deep-typed client improves to 5x.

5.3.3 Deep or Shallow, Worst-Case. Both deep and shallow implementations have known bottlenecks. With deep types, high-traffic boundaries can lead to huge costs [35, 57, 30]. With shallow types, every line of typed code contributes a small cost [70, 29].

By switching between Deep and Shallow a programmer can often, however, avoid the worst-cases of each. Figure 16 quantifies the benefits of this either-or strategy on the GTP benchmarks. The first column shows that, as expected, deep types may have enormous costs. The second column shows that the worst configurations for Shallow Racket are far less severe. The third column shows, however, that toggling between Deep and Shallow often avoids the pathologies of each style. Numbers in this third column are typeset in **bold** if they are the best (lowest) in their row.

Benchmark	Worst Deep	Worst Shallow	Worst D||S
sieve	16x	4.36x	**2.97x**
forth	5800x	5.51x	**5.43x**
fsm	2.24x	2.38x	**1.91x**
fsmoo	420x	4.28x	**4.25x**
mbta	1.91x	1.74x	**1.71x**
morsecode	1.57x	2.77x	**1.3x**
zombie	46x	31x	31x
dungeon	15000x	4.97x	**3.16x**
jpeg	23x	1.66x	**1.56x**
zordoz	2.63x	2.75x	**2.58x**
lnm	1.23x	1.21x	**1.17x**
suffixtree	31x	5.8x	5.8x
kcfa	4.33x	1.24x	1.24x
snake	12x	7.67x	**7.61x**
take5	44x	2.99x	**2.97x**
acquire	4.22x	1.42x	1.42x
tetris	13x	9.93x	**5.44x**
synth	47x	4.2x	4.2x
gregor	1.72x	1.59x	**1.51x**
quadT	26x	7.39x	**7.23x**
quadU	55x	7.57x	**7.45x**

Figure 16: Worst-case overheads vs. the untyped configuration for Deep alone, Shallow alone, and an either-or mix.

Remark: the either-or "toggling" strategy is possible only because Deep and Shallow can interoperate. Most of the benchmarks rely on deep-typed code that lives outside their N core migratable modules (16 out of 21 benchmarks). Without interoperability, the outside code would require changes that are unrealistic to make in practice.

In figure 16, the `sieve` and `tetris` benchmarks are notable successes. The `zombie` benchmark is the worst. Deep Racket pays a huge cost in `zombie` because functions repeatedly cross its module boundaries. Shallow Racket pays a high cost as well because `zombie` uses functions to simulate message-passing objects, and therefore contains many elimination forms that incur shape checks.

5.3.4 Migration Paths. The complementary strengths of Deep and Shallow Racket can help programmers avoid bottlenecks as they migrate an untyped codebase to a typed configuration. Consider the set of all migration paths, each of which begins at the untyped configuration and adds types to one module at a time until reaching the fully-typed configuration. A path is D-deliverable if all of its configurations run at most D times slower than the untyped configuration.

Figure 17 counts the proportion of 3-deliverable paths out of all $N!$ migration paths in a subset of the GTP benchmarks. Larger benchmarks are omitted. The first column counts paths in Deep Racket, the second column counts paths in Shallow Racket, and the third column counts paths using Deep or Shallow at each point. With Deep alone, all paths

[4] `gitlab.com/HiPhish/MsgPack.rkt`

[5] `github.com/stamourv/synth`

Benchmark	Deep paths	Shallow paths	D‖S paths
sieve	0%	0%	**100%**
forth	0%	0%	**50%**
fsm	100%	100%	100%
fsmoo	0%	0%	**50%**
mbta	100%	100%	100%
morsecode	100%	100%	100%
zombie	0%	0%	**50%**
dungeon	0%	0%	**67%**
jpeg	0%	100%	100%
zordoz	100%	100%	100%
lnm	100%	100%	100%
suffixtree	0%	0%	**12%**
kcfa	33%	100%	100%
snake	0%	0%	0%
take5	0%	100%	100%

Figure 17: Percent of 3-deliverable migration paths for Deep alone, Shallow alone, and an either-or mix.

in nine benchmarks reach a bottleneck that exceeds the 3x limit. With Shallow alone, all paths in seven benchmarks exceed the limit as well—often near the end of the migration path. With the either-or mix, only one benchmark (snake) has zero 3-deliverable paths.

6 Related Work

Two gradual languages, Thorn [72] and StrongScript [50], support a combination of sound *concrete* types and erased *like* types. Thorn is a scalable scripting language that compiles to the JVM [7]. StrongScript extends TypeScript [6] with concrete types. Pyret explores a type-based combination, with deep checks for types that describe fixed-size data and shallow checks for other types.[6] For example, pair types get a deep check and list types get a shallow check. Static Python combines shallow and concrete checks [4]. Shallow checks are the default, and programmers can opt-in to concrete data structures. Outside the realm of gradual typing, option contracts allow client code to trust (and skip checking) specific contracts from server code [11].

The model in section 3 builds on the semantic framework of Greenman and Felleisen [26], which is in turn inspired by Matthews and Findler [40]. Unlike those frameworks, the present model uses a surface-to-evaluation compiler similar to how Chung et al. [9] compile several gradual languages to the KafKa core language. The compiler in section 3 is inspired by the coercion calculus [34]; in particular, its *completion* pass that makes run-time type checks explicit.

There is a great deal of related work that addresses the performance of deep or shallow types via implementation techniques [15], static analysis [47, 46, 69, 43], compilation techniques [5, 49, 51], and clean-slate language designs [44, 38, 45]. These improvements are orthogonal to a combined language; they should apply to a three-way language as well as any normal gradual language. As a case in point, our three-way Typed Racket benefits from collapsible contracts [15].

7 Future Work

One drawback apparent in the model is that deep and shallow cannot trust one another. Deep code always wraps inputs from shallow code because they may have originated in untyped code. Greenman [23] sketches two ideas for removing checks from deep–shallow boundaries. One requires an escape analysis. The other asks for a shallow semantics that creates wrappers (such as in [8]) instead of the transient semantics. A third idea is to adapt confined gradual typing [1]. If the type system can prove that confined values originate in typed code and never escape to untyped, then deep and shallow can freely share these values.

A second future direction is to identify best practices for coding in a three-way language. Anecdotal experience suggests the following strategy:

1. Start by adding deep types because their strong guarantees may help identify logical errors.
2. If performance becomes an issue, switch to shallow.
3. Once all critical boundaries are typed, use deep to maximize the effect of type-driven optimizations.

Adapting the notion of a *rational programmer* [39] may provide a way to systematically test the usefulness of this migration plan. Meanwhile, there may be additional ways to leverage the spectrum of type enforcement.

8 Conclusion

This is the first implementation of a sound gradual type system where programmers can explicitly choose to trade performance for guarantees as they add types. If a new set of type annotations brings unacceptable overhead, switching the types' semantics from deep to shallow can avoid the bottleneck and may even be good enough to deploy. The guarantees from deep types can always be used for debugging the inevitable failure, and can be applied sparingly to defend a critical module. In the future, implementors may wish to explore other ways to trade performance for guarantees, making the trade-off even more programmable.

Data Availability Statement

The datasets and software that support section 5 of this paper are available via Software Heritage [24] and Zenodo [25].

Acknowledgments

This work is supported by NSF grant 1518844, NSF grant 1763922, and NSF grant 2030859 to the CRA for the CIFellows project. Thanks to Matthias Felleisen for improving drafts of this paper and to the rest of my thesis committee

[6] Personal communication. `pyret.org`

for supervising parts of this work: Amal Ahmed, Fritz Henglein, Shriram Krishnamurthi, Sam Tobin-Hochstadt, and Jan Vitek.

A Appendix

A.1 Surface Typing, Completion, Evaluation Typing, and Label Consistency Judgments

Figure 18 presents the full typing judgment for the surface language. Figure 19 presents the surface-to-evaluation compilation rules. Three other figures present the evaluation typing judgments: figure 20 for deep types, figure 21 for shallow types, and figure 22 for dynamic typing. Figure #f presents the consistency judgment for labeled expressions.

A.2 How to Lift a Reduction Relation

A.2.1 Background on Complete Monitoring. The idea behind complete monitoring is to test whether a semantics has control over every interaction between typed and untyped code. If the property holds, then a programmer can rely on the language to insert checks at the right places, for example, between the library and client demonstrated in figure 1. As a concrete example, if a value passes through the type $(\mathsf{Int} \rightarrow \mathsf{Int})$ then complete monitoring guarantees that the language has control over every input to the function and every result that the function computes, regardless of what context these interactions occur in.

Because all such interactions originate at the boundaries between typed and untyped code, a very simple way to formalize complete monitoring is to ask whether each boundary comes with a full run-time check when possible and an error otherwise. A language that meets this strict requirement certainly has full control. Other good designs fail, though. Suppose typed code expects a pair of integers and a semantics initially admits any pair at the boundary but eventually checks that the pair contains integers. Despite the incomplete check at the boundary, this delayed-checking semantics eventually performs all necessary checks and should satisfy a complete monitoring theorem. Higher-order values raise a similar question because a single run-time check cannot prove that a function value always behaves a certain way. Nevertheless, a language that checks every call and return is in full control of the function's interactions.

Our definition of complete monitoring translates these ideas about interactions and control into statements about *ownership labels* [12]. At the start of an evaluation, no interactions have occurred yet and every expression has one owner: the enclosing component. The reduction of a boundary term is the semantics of an interaction in which a value flows from one sender component to a client. At this point, the sender loses full control over the value. If the value fully matches the type expectations of the client, then the loss of control is no problem and the client gains full ownership. Otherwise, the sender and client may have to assume joint ownership of

$$\boxed{\Gamma \vdash_s s : T}$$

$$\frac{(x_0 : \mathcal{U}) \in \Gamma}{\Gamma \vdash_s x_0 : \mathcal{U}} \qquad \frac{(x_0 : \tau_0) \in \Gamma}{\Gamma \vdash_s x_0 : \tau_0} \qquad \frac{(x_0 : \llcorner\tau_0\lrcorner) \in \Gamma}{\Gamma \vdash_s x_0 : \llcorner\tau_0\lrcorner}$$

$$\frac{}{\Gamma \vdash_s i_0 : \mathcal{U}} \qquad \frac{}{\Gamma \vdash_s n_0 : \mathsf{Nat}} \qquad \frac{}{\Gamma \vdash_s n_0 : \llcorner\mathsf{Nat}\lrcorner}$$

$$\frac{}{\Gamma \vdash_s i_0 : \mathsf{Int}} \qquad \frac{}{\Gamma \vdash_s i_0 : \llcorner\mathsf{Int}\lrcorner} \qquad \frac{\Gamma \vdash_s s_0 : \mathcal{U} \quad \Gamma \vdash_s s_1 : \mathcal{U}}{\Gamma \vdash_s \langle s_0, s_1 \rangle : \mathcal{U}}$$

$$\frac{\Gamma \vdash_s s_0 : \tau_0 \quad \Gamma \vdash_s s_1 : \tau_1}{\Gamma \vdash_s \langle s_0, s_1 \rangle : \tau_0 \times \tau_1} \qquad \frac{\Gamma \vdash_s s_0 : \llcorner\tau_0\lrcorner \quad \Gamma \vdash_s s_1 : \llcorner\tau_1\lrcorner}{\Gamma \vdash_s \langle s_0, s_1 \rangle : \llcorner\tau_0 \times \tau_1\lrcorner}$$

$$\frac{(x_0 : \mathcal{U}), \Gamma \vdash_s e_0 : \mathcal{U}}{\Gamma \vdash_s \lambda x_0.\, e_0 : \mathcal{U}} \qquad \frac{(x_0 : \tau_0), \Gamma \vdash_s e_0 : \tau_1}{\Gamma \vdash_s \lambda(x_0 : \tau_0).\, e_0 : \tau_0 \rightarrow \tau_1}$$

$$\frac{(x_0 : \llcorner\tau_0\lrcorner), \Gamma \vdash_s e_0 : \llcorner\tau_1\lrcorner}{\Gamma \vdash_s \lambda(x_0 : \tau_0).\, e_0 : \llcorner\tau_0 \rightarrow \tau_1\lrcorner} \qquad \frac{\Gamma \vdash_s s_0 : \mathcal{U}}{\Gamma \vdash_s \mathit{unop}\, s_0 : \mathcal{U}}$$

$$\frac{\begin{array}{c}\Gamma \vdash_s e_0 : \tau_0 \\ \Delta(\mathit{unop}, \tau_0) = \tau_1\end{array}}{\Gamma \vdash_s \mathit{unop}\, e_0 : \tau_1} \qquad \frac{\Gamma \vdash_s e_0 : \llcorner\tau_0\lrcorner \quad \Delta(\mathit{unop}, \tau_0) = \tau_1}{\Gamma \vdash_s \mathit{unop}\, e_0 : \llcorner\tau_1\lrcorner}$$

$$\frac{\Gamma \vdash_s s_0 : \mathcal{U} \quad \Gamma \vdash_s s_1 : \mathcal{U}}{\Gamma \vdash_s \mathit{binop}\, s_0\, s_1 : \mathcal{U}} \qquad \frac{\begin{array}{c}\Gamma \vdash_s e_0 : \tau_0 \quad \Gamma \vdash_s e_1 : \tau_1 \\ \Delta(\mathit{binop}, \tau_0, \tau_1) = \tau_2\end{array}}{\Gamma \vdash_s \mathit{binop}\, e_0\, e_1 : \tau_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash_s e_0 : \llcorner\tau_0\lrcorner \quad \Gamma \vdash_s e_1 : \llcorner\tau_1\lrcorner \\ \Delta(\mathit{binop}, \tau_0, \tau_1) = \tau_2\end{array}}{\Gamma \vdash_s \mathit{binop}\, e_0\, e_1 : \llcorner\tau_2\lrcorner} \qquad \frac{\Gamma \vdash_s s_0 : \mathcal{U} \quad \Gamma \vdash_s s_1 : \mathcal{U}}{\Gamma \vdash_s \mathsf{app}\{s_0\}\, s_1 : \mathcal{U}}$$

$$\frac{\begin{array}{c}\Gamma \vdash_s e_0 : \tau_0 \rightarrow \tau_1 \\ \Gamma \vdash_s e_1 : \tau_0\end{array}}{\Gamma \vdash_s \mathsf{app}\{e_0\}\, e_1 : \tau_1} \qquad \frac{\begin{array}{c}\Gamma \vdash_s e_0 : \llcorner\tau_0 \rightarrow \tau_1\lrcorner \\ \Gamma \vdash_s e_1 : \llcorner\tau_0\lrcorner\end{array}}{\Gamma \vdash_s \mathsf{app}\{e_0\}\, e_1 : \llcorner\tau_1\lrcorner}$$

$$\frac{\Gamma \vdash_s e_0 : \tau_0 \quad \tau_0 <: \tau_1}{\Gamma \vdash_s e_0 : \tau_1} \qquad \frac{\Gamma \vdash_s e_0 : \llcorner\tau_0\lrcorner \quad \tau_0 <: \tau_1}{\Gamma \vdash_s e_0 : \llcorner\tau_1\lrcorner}$$

$$\left[\frac{\Gamma \vdash_s e_0 : T_0}{\Gamma \vdash_s \mathsf{module}\, L_0\, e_0 : T_1}\right]$$

L_0	T_0	T_1
$\mathcal{D}$	τ_0	τ_0
$\mathcal{D}$	τ_0	$\llcorner\tau_0\lrcorner$
$\mathcal{D}$	τ_0	$\mathcal{U}$
$\mathcal{S}$	$\llcorner\tau_0\lrcorner$	τ_0
$\mathcal{S}$	$\llcorner\tau_0\lrcorner$	$\llcorner\tau_0\lrcorner$
$\mathcal{S}$	$\llcorner\tau_0\lrcorner$	$\mathcal{U}$
$\mathcal{U}$	$\mathcal{U}$	T_1

Figure 18: Surface typing judgment

$$\frac{}{\Gamma \vdash_{\mathrm{s}} x_0 : \mathcal{U} \rightsquigarrow x_0} \qquad \frac{}{\Gamma \vdash_{\mathrm{s}} x_0 : \tau_0 \rightsquigarrow x_0}$$

$$\frac{}{\Gamma \vdash_{\mathrm{s}} x_0 : \lfloor \tau_0 \rfloor \rightsquigarrow x_0} \qquad \frac{}{\Gamma \vdash_{\mathrm{s}} i_0 : \mathcal{U} \rightsquigarrow i_0}$$

$$\frac{}{\Gamma \vdash_{\mathrm{s}} i_0 : \tau_0 \rightsquigarrow i_0} \qquad \frac{}{\Gamma \vdash_{\mathrm{s}} i_0 : \lfloor \tau_0 \rfloor \rightsquigarrow i_0}$$

$$\frac{\Gamma \vdash_{\mathrm{s}} e_0 : \mathcal{U} \rightsquigarrow e_2 \quad \Gamma \vdash_{\mathrm{s}} e_1 : \mathcal{U} \rightsquigarrow e_3}{\Gamma \vdash_{\mathrm{s}} \langle e_0, e_1 \rangle : \mathcal{U} \rightsquigarrow \langle e_2, e_3 \rangle}$$

$$\frac{\Gamma \vdash_{\mathrm{s}} e_0 : \tau_0 \rightsquigarrow e_2 \quad \Gamma \vdash_{\mathrm{s}} e_1 : \tau_1 \rightsquigarrow e_3}{\Gamma \vdash_{\mathrm{s}} \langle e_0, e_1 \rangle : \tau_0 \times \tau_1 \rightsquigarrow \langle e_2, e_3 \rangle}$$

$$\frac{\Gamma \vdash_{\mathrm{s}} e_0 : \lfloor \tau_0 \rfloor \rightsquigarrow e_2 \quad \Gamma \vdash_{\mathrm{s}} e_1 : \lfloor \tau_1 \rfloor \rightsquigarrow e_3}{\Gamma \vdash_{\mathrm{s}} \langle e_0, e_1 \rangle : \lfloor \tau_0 \times \tau_1 \rfloor \rightsquigarrow \langle e_2, e_3 \rangle}$$

$$\frac{(x_0 : \mathcal{U}), \Gamma \vdash_{\mathrm{s}} e_0 : \mathcal{U} \rightsquigarrow e_1}{\Gamma \vdash_{\mathrm{s}} \lambda x_0.\, e_0 : \mathcal{U} \rightsquigarrow \lambda x_0.\, e_1}$$

$$\frac{(x_0 : \tau_0), \Gamma \vdash_{\mathrm{s}} e_0 : \tau_1 \rightsquigarrow e_1}{\Gamma \vdash_{\mathrm{s}} \lambda(x_0 : \tau_0).\, e_0 : \tau_0 \to \tau_1 \rightsquigarrow \lambda(x_0 : \tau_0).\, e_1}$$

$$\frac{(x_0 : \lfloor \tau_0 \rfloor), \Gamma \vdash_{\mathrm{s}} e_0 : \lfloor \tau_1 \rfloor \rightsquigarrow e_1 \quad \mathit{shape}(\tau_0) = \sigma_0}{\Gamma \vdash_{\mathrm{s}} \lambda(x_0 : \lfloor \tau_0 \rfloor).\, e_0 : \lfloor \tau_0 \to \tau_1 \rfloor \rightsquigarrow \lambda(x_0 : \mathsf{scan}\, \sigma_0).\, e_1}$$

$$\frac{\Gamma \vdash_{\mathrm{s}} e_0 : \mathcal{U} \rightsquigarrow e_2 \quad \Gamma \vdash_{\mathrm{s}} e_1 : \mathcal{U} \rightsquigarrow e_3}{\Gamma \vdash_{\mathrm{s}} \mathsf{app}\, e_0\, e_1 : \mathcal{U} \rightsquigarrow \mathsf{app}\, e_2\, e_3}$$

$$\frac{\Gamma \vdash_{\mathrm{s}} e_0 : \tau_1 \to \tau_0 \rightsquigarrow e_2 \quad \Gamma \vdash_{\mathrm{s}} e_1 : \tau_1 \rightsquigarrow e_3}{\Gamma \vdash_{\mathrm{s}} \mathsf{app}\, e_0\, e_1 : \tau_0 \rightsquigarrow \mathsf{app}\, e_2\, e_3}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{s}} e_0 : \lfloor \tau_1 \to \tau_0 \rfloor \rightsquigarrow e_2 \\ \Gamma \vdash_{\mathrm{s}} e_1 : \lfloor \tau_1 \rfloor \rightsquigarrow e_3 \quad \mathit{shape}(\tau_0) = \sigma_0\end{array}}{\Gamma \vdash_{\mathrm{s}} \mathsf{app}\, e_0\, e_1 : \lfloor \tau_0 \rfloor \rightsquigarrow \mathsf{scan}\, \sigma_0\, (\mathsf{app}\, e_2\, e_3)}$$

$$\left[\frac{\Gamma \vdash_{\mathrm{s}} e_0 : T_0 \rightsquigarrow e_1}{\Gamma \vdash_{\mathrm{s}} \mathsf{module}\, L_0\ e_0 : T_1 \rightsquigarrow e_2}\right] \quad \begin{array}{llll} L_0 & T_0 & T_1 & \rightsquigarrow e_2 \\ \hline \mathcal{D} & \tau_0 & \tau_0 & \mathsf{noop}\ e_1 \\ \mathcal{S} & \lfloor \tau_0 \rfloor & \lfloor \tau_0 \rfloor & \mathsf{noop}\ e_1 \\ \mathcal{U} & \mathcal{U} & \mathcal{U} & \mathsf{noop}\ e_1 \\ \mathcal{D} & \tau_0 & \mathcal{U} & \mathsf{wrap}\, \tau_0\ e_1 \\ \mathcal{S} & \lfloor \tau_0 \rfloor & \tau_0 & \mathsf{wrap}\, \tau_0\ e_1 \\ \mathcal{U} & \mathcal{U} & \tau_0 & \mathsf{wrap}\, \tau_0\ e_1 \\ \mathcal{D} & \tau_0 & \lfloor \tau_0 \rfloor & \mathsf{wrap}\, \tau_0\ e_1 \\ \mathcal{S} & \lfloor \tau_0 \rfloor & \mathcal{U} & \mathsf{noop}\ e_1 \\ \mathcal{U} & \mathcal{U} & \lfloor \tau_0 \rfloor & \mathsf{scan}\, \sigma_0\ e_1 \\ \multicolumn{4}{r}{\text{where } \sigma_0 = \mathit{shape}(\tau_0)} \end{array}$$

Figure 19: Surface-to-evaluation compilation

$$\boxed{\Gamma \vdash_{\mathcal{D}} e : \tau}$$

$$\frac{(x_0 : \tau_0) \in \Gamma}{\Gamma \vdash_{\mathcal{D}} x_0 : \tau_0} \qquad \frac{}{\Gamma \vdash_{\mathcal{D}} n_0 : \mathsf{Nat}} \qquad \frac{}{\Gamma \vdash_{\mathcal{D}} i_0 : \mathsf{Int}}$$

$$\frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0 \quad \Gamma \vdash_{\mathcal{D}} e_1 : \tau_1}{\Gamma \vdash_{\mathcal{D}} \langle e_0, e_1 \rangle : \tau_0 \times \tau_1} \qquad \frac{(x_0 : \tau_0), \Gamma \vdash_{\mathcal{D}} e_0 : \tau_1}{\Gamma \vdash_{\mathcal{D}} \lambda(x_0 : \tau_0).\, e_0 : \tau_0 \to \tau_1}$$

$$\frac{\Gamma \vdash_{\mathcal{U}} v_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{D}} \mathbb{G}\, \tau_0\, v_0 : \tau_0} \qquad \frac{\Gamma \vdash_{\mathcal{S}} v_0 : \sigma_0}{\Gamma \vdash_{\mathcal{D}} \mathbb{G}\, \tau_0\, v_0 : \tau_0}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0 \\ \Delta(\mathit{unop}, \tau_0) = \tau_1\end{array}}{\Gamma \vdash_{\mathcal{D}} \mathit{unop}\, e_0 : \tau_1} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0 \quad \Gamma \vdash_{\mathcal{D}} e_1 : \tau_1 \\ \Delta(\mathit{binop}, \tau_0, \tau_1) = \tau_2\end{array}}{\Gamma \vdash_{\mathcal{D}} \mathit{binop}\, e_0\, e_1 : \tau_2}$$

$$\frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0 \to \tau_1 \quad \Gamma \vdash_{\mathcal{D}} e_1 : \tau_0}{\Gamma \vdash_{\mathcal{D}} \mathsf{app}\, e_0\, e_1 : \tau_1} \qquad \frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0}{\Gamma \vdash_{\mathcal{D}} \mathsf{noop}\ e_0 : \tau_0}$$

$$\frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{D}} \mathsf{wrap}\, \tau_0\ e_0 : \tau_0} \qquad \frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_0}{\Gamma \vdash_{\mathcal{D}} \mathsf{wrap}\, \tau_0\ e_0 : \tau_0}$$

$$\frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0 \quad \tau_0 <: \tau_1}{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_1} \qquad \frac{}{\Gamma \vdash_{\mathcal{D}} \mathsf{Error} : \tau_0}$$

Figure 20: Deep typing judgment

the value depending on the nature of the reduction relation. If a semantics can create a value with multiple owners, then it admits that a component may lose full control over its interactions with other components.

Technically, an ownership label $^{\ell_0}$ names one source-code component. Expressions and values come with at least one ownership label; for example, $(42)^{\ell_0}$ is an integer with one owner and $(((42)^{\ell_0})^{\ell_1})^{\ell_2}$ is an integer with three owners, written $(\!(42)\!)^{\ell_0\ell_1\ell_2}$ for short. A complete monitoring theorem requires two ingredients that manage these labels. First, a reduction relation $\to_{\mathrm{r}}^{*}$ must propagate ownership labels to reflect interactions and checks. Second, a single-ownership judgment $\Vdash$ must test whether every value in an expression has a unique owner. To satisfy complete monitoring, reduction must preserve single-ownership.

The key single-ownership rules (figure #f) deal with labeled expressions and boundary terms. For example, every label in a deep-typed component must match a (possibly different) deep-typed component, and the subterm in every scan boundary must match its server label:

$$\boxed{\mathcal{L}; \ell \Vdash e} \qquad \frac{\mathcal{D}_1; \mathcal{L}_0 \Vdash e_0}{\mathcal{D}_0; \mathcal{L}_0 \Vdash (e_0)^{\mathcal{D}_1}} \quad \frac{\ell_1; \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \mathsf{scan}\, \sigma_0\ (e_0)^{\ell_1}}$$

Values such as $(\!(42)\!)^{\ell_0\ell_1}$ represent a communication that slipped past the run-time checking protocol, and therefore

$$\boxed{\Gamma \vdash_{\mathcal{S}} e : \sigma}$$

$$\frac{(x_0 : \sigma_0) \in \Gamma}{\Gamma \vdash_{\mathcal{S}} x_0 : \sigma_0} \qquad \frac{}{\Gamma \vdash_{\mathcal{S}} n_0 : \mathsf{Nat}} \qquad \frac{}{\Gamma \vdash_{\mathcal{S}} i_0 : \mathsf{Int}}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_0 \quad \Gamma \vdash_{\mathcal{S}} e_1 : \sigma_1}{\Gamma \vdash_{\mathcal{S}} \langle e_0, e_1 \rangle : \mathsf{Pair}} \qquad \frac{(x_0 : \mathcal{U}), \Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{S}} \lambda x_0.\, e_0 : \mathsf{Fun}}$$

$$\frac{(x_0 : \sigma_0), \Gamma \vdash_{\mathcal{S}} e_0 : \sigma_1}{\Gamma \vdash_{\mathcal{S}} \lambda(x_0 : \mathsf{scan}\, \sigma_0).\, e_0 : \mathsf{Fun}} \qquad \frac{\Gamma \vdash_{\mathcal{D}} v_0 : \tau_0}{\Gamma \vdash_{\mathcal{S}} \mathbb{G}\, \tau_0\, v_0 : \mathsf{Fun}}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_0}{\Gamma \vdash_{\mathcal{S}} \mathit{unop}\, e_0 : \mathsf{Top}} \qquad \frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_0 \quad \Gamma \vdash_{\mathcal{S}} e_1 : \sigma_1 \\ \Delta(\mathit{binop}, \sigma_0, \sigma_1) = \sigma_2}{\Gamma \vdash_{\mathcal{S}} \mathit{binop}\, e_0\, e_1 : \sigma_2}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_0 : \mathsf{Fun} \quad \Gamma \vdash_{\mathcal{S}} e_1 : \sigma_0}{\Gamma \vdash_{\mathcal{S}} \mathsf{app}\, e_0\, e_1 : \mathsf{Top}} \qquad \frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_0}{\Gamma \vdash_{\mathcal{S}} \mathsf{noop}\ e_0 : \sigma_0}$$

$$\frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{S}} \mathsf{noop}\ e_0 : \mathsf{Top}} \qquad \frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{S}} \mathsf{scan}\, \sigma_0\ e_0 : \sigma_0}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_1}{\Gamma \vdash_{\mathcal{S}} \mathsf{scan}\, \sigma_0\ e_0 : \sigma_0} \qquad \frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0 \quad \mathit{shape}(\tau_0) = \sigma_0}{\Gamma \vdash_{\mathcal{S}} \mathsf{wrap}\, \tau_0\ e_0 : \sigma_0}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_0 \quad \sigma_0 <: \sigma_1}{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_1} \qquad \frac{}{\Gamma \vdash_{\mathcal{S}} \mathsf{Error} : \sigma_0}$$

Figure 21: Shallow typing judgment

fail to satisfy single ownership. The client owns the wrapper, and the sender retains ownership of the enclosed value.

A.2.2 Lifting a Reduction Relation. In practice, a language comes with an unlabeled reduction relation, and it is up to a researcher to design a lifted relation that handles labeled terms. Lifting requires insight to correctly transfer labels and to ensure that labels do not change the behavior of programs. If labels do not transfer correctly, then a complete monitoring theorem becomes meaningless. And if the lifted relation depends on labels to compute a result, then a complete monitoring theorem says nothing about the original reduction relation.

The following informal guidelines explain how to lift a reduction relation. They convey the intuitions behind our formulation of complete monitoring and those of prior work [12, 13, 58, 42]. Each guideline describes a way that labels may be transferred or dropped during evaluation. To convey the general idea, each guideline also comes with a brief illustration, namely, an example reduction and a short comment. The example reductions use a hypothetical $\mathbf{r}$ relation over the surface language. Recall that stat and dyn are boundary terms; they link two components, a context and an enclosed expression, via a type.

$$\boxed{\Gamma \vdash_{\mathcal{U}} e : \mathcal{U}}$$

$$\frac{(x_0 : \mathcal{U}) \in \Gamma}{\Gamma \vdash_{\mathcal{U}} x_0 : \mathcal{U}} \qquad \frac{}{\Gamma \vdash_{\mathcal{U}} i_0 : \mathcal{U}}$$

$$\frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U} \quad \Gamma \vdash_{\mathcal{U}} e_1 : \mathcal{U}}{\Gamma \vdash_{\mathcal{U}} \langle e_0, e_1 \rangle : \mathcal{U}} \qquad \frac{(x_0 : \mathcal{U}), \Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{U}} \lambda x_0.\, e_0 : \mathcal{U}}$$

$$\frac{(x_0 : \sigma_0), \Gamma \vdash_{\mathcal{S}} e_0 : \sigma_1}{\Gamma \vdash_{\mathcal{U}} \lambda(x_0 : \mathsf{scan}\, \sigma_0).\, e_0 : \mathcal{U}} \qquad \frac{\Gamma \vdash_{\mathcal{D}} v_0 : \tau_0}{\Gamma \vdash_{\mathcal{U}} \mathbb{G}\, \tau_0\, v_0 : \mathcal{U}}$$

$$\frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{U}} \mathit{unop}\, e_0 : \mathcal{U}} \qquad \frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U} \quad \Gamma \vdash_{\mathcal{U}} e_1 : \mathcal{U}}{\Gamma \vdash_{\mathcal{U}} \mathit{binop}\, e_0\, e_1 : \mathcal{U}}$$

$$\frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U} \quad \Gamma \vdash_{\mathcal{U}} e_1 : \mathcal{U}}{\Gamma \vdash_{\mathcal{U}} \mathsf{app}\, e_0\, e_1 : \mathcal{U}} \qquad \frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{U}} \mathsf{noop}\ e_0 : \mathcal{U}}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_0}{\Gamma \vdash_{\mathcal{U}} \mathsf{noop}\ e_0 : \mathcal{U}} \qquad \frac{\Gamma \vdash_{\mathcal{U}} e_0 : \mathcal{U}}{\Gamma \vdash_{\mathcal{U}} \mathsf{scan}\, \sigma_0\ e_0 : \mathcal{U}}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_0 : \sigma_1}{\Gamma \vdash_{\mathcal{U}} \mathsf{scan}\, \sigma_0\ e_0 : \mathcal{U}} \qquad \frac{\Gamma \vdash_{\mathcal{D}} e_0 : \tau_0}{\Gamma \vdash_{\mathcal{U}} \mathsf{wrap}\, \tau_0\ e_0 : \mathcal{U}}$$

$$\frac{}{\Gamma \vdash_{\mathcal{U}} \mathsf{Error} : \mathcal{U}}$$

Figure 22: Untyped typing judgment

When reading an example, accept the transitions $e\ \mathbf{r}\ e$ as axioms and focus on how the labels change in response:

1. If a base value reaches a boundary with a matching base type, then the value must drop its current labels as it crosses the boundary.
   $$(\mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (\!(0)\!)^{\ell_2 \ell_1})^{\ell_0}\ \ \mathbf{r}\ \ (0)^{\ell_0}$$
   The value 0 *fully matches the type* Nat.
2. Any other value that crosses a boundary must acquire the label of the new context.
   $$(\mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (\langle -2, 1 \rangle)^{\ell_1})^{\ell_0}\ \ \mathbf{r}\ \ (\!(\langle -2, 1 \rangle)\!)^{\ell_1 \ell_0}$$
   The pair $\langle -2, 1 \rangle$ *does not match the type* Nat.
3. Every value that flows out of a value v_0 acquires the labels of v_0 and the context.
   $$(\mathsf{snd}\ (\!(\langle (1)^{\ell_0}, (2)^{\ell_1} \rangle)\!)^{\ell_2 \ell_3})^{\ell_4}\ \ \mathbf{r}\ \ (\!(2)\!)^{\ell_1 \ell_2 \ell_3 \ell_4}$$
   The value 2 *flows out of the pair* $\langle 1, 2 \rangle$.
4. Every value that flows into a function v_0 acquires the label of the context and the reversed labels of v_0.
   $$(\mathsf{app}\ (\!(\lambda x_0.\, \mathsf{fst}\ x_0)\!)^{\ell_0 \ell_1}\ (\langle 8, 6 \rangle)^{\ell_2})^{\ell_3}\ \ \mathbf{r}$$
   $$(\!(\mathsf{fst}\ (\!(\langle 8, 6 \rangle)\!)^{\ell_2 \ell_3 \ell_1 \ell_0})\!)^{\ell_0 \ell_1})^{\ell_3}$$
   The argument value $\langle 8, 6 \rangle$ *is input to the function.*

$$\boxed{\ell; \mathcal{L} \Vdash e}$$

$$\dfrac{(x_0 : \ell_0) \in \mathcal{L}_0}{\ell_0; \mathcal{L}_0 \Vdash x_0} \qquad \dfrac{}{\ell_0; \mathcal{L}_0 \Vdash i_0} \qquad \dfrac{\ell_0; \mathcal{L}_0 \Vdash e_0 \quad \ell_0; \mathcal{L}_0 \Vdash e_1}{\ell_0; \mathcal{L}_0 \Vdash \langle e_0, e_1 \rangle}$$

$$\dfrac{\ell_0; (x_0 : \ell_0), \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \lambda(x_0 : \tau_0).\, e_0} \qquad \dfrac{\ell_0; (x_0 : \ell_0), \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \lambda(x_0 : \mathsf{scan}\, \sigma_0).\, e_0}$$

$$\dfrac{\ell_0; (x_0 : \ell_0), \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \lambda x_0.\, e_0} \qquad \dfrac{\ell_0; \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \mathit{unop}\, e_0}$$

$$\dfrac{\ell_0; \mathcal{L}_0 \Vdash e_0 \quad \ell_0; \mathcal{L}_0 \Vdash e_1}{\ell_0; \mathcal{L}_0 \Vdash \mathit{binop}\, e_0\, e_1} \qquad \dfrac{\ell_0; \mathcal{L}_0 \Vdash e_0 \quad \ell_0; \mathcal{L}_0 \Vdash e_1}{\ell_0; \mathcal{L}_0 \Vdash \mathsf{app}\, e_0\, e_1}$$

$$\dfrac{}{\ell_0; \mathcal{L}_0 \Vdash \mathsf{Error}} \qquad \dfrac{\ell_1; \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \mathsf{noop}\ (e_0)^{\ell_1}}$$

$$\dfrac{\ell_1; \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \mathsf{scan}\, \sigma_0\ (e_0)^{\ell_1}} \qquad \dfrac{\ell_1; \mathcal{L}_0 \Vdash e_0}{\ell_0; \mathcal{L}_0 \Vdash \mathsf{wrap}\, \tau_0\ (e_0)^{\ell_1}}$$

$$\dfrac{\ell_1; \mathcal{L}_0 \Vdash v_0}{\ell_0; \mathcal{L}_0 \Vdash \mathbb{G}\, \tau_0\ (v_0)^{\ell_1}} \qquad \dfrac{\mathcal{D}_1; \mathcal{L}_0 \Vdash e_0}{\mathcal{D}_0; \mathcal{L}_0 \Vdash (e_0)^{\mathcal{D}_1}} \qquad \dfrac{\mathcal{S}_1; \mathcal{L}_0 \Vdash e_0}{\mathcal{S}_0; \mathcal{L}_0 \Vdash (e_0)^{\mathcal{S}_1}}$$

$$\dfrac{\mathcal{U}_0; \mathcal{L}_0 \Vdash e_0}{\mathcal{S}_0; \mathcal{L}_0 \Vdash (e_0)^{\mathcal{U}_0}} \qquad \dfrac{\mathcal{U}_1; \mathcal{L}_0 \Vdash e_0}{\mathcal{U}_0; \mathcal{L}_0 \Vdash (e_0)^{\mathcal{U}_1}} \qquad \dfrac{\mathcal{S}_0; \mathcal{L}_0 \Vdash e_0}{\mathcal{U}_0; \mathcal{L}_0 \Vdash (e_0)^{\mathcal{S}_0}}$$

Figure 23: Deep label consistency

The substituted body flows out of the function,
and by guideline 3 acquires the function's labels.

5. A primitive operation (δ) may remove labels on incoming base values.

   $$(\mathsf{plus}\ (2)^{\ell_0}\ (3)^{\ell_1})^{\ell_2}\ \mathbf{r}\ (5)^{\ell_2}$$
   Assuming $\delta(\mathsf{plus}, 2, 3) = 5$.

6. Consecutive equal labels may be dropped.

   $$(\!(0)\!)^{\ell_0\ell_0\ell_1\ell_0} = (\!(0)\!)^{\ell_0\ell_1\ell_0}$$

7. Labels on an error term may be dropped.

   $$(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_1)\ (\mathsf{plus}\ 9\ (\mathsf{DivErr})^{\ell_1}))^{\ell_0}\ \mathbf{r}\ \ \mathsf{DivErr}$$

Although guideline 4 talks about functions, it generalizes to reference cells and other higher-order values.

A.3 Lemmas for the Model

Theorem A.1 (complete monitoring). *If* $\vdash_s s_0 : T \rightsquigarrow e_0$ *and* $\ell_0; \cdot \Vdash e_0$ *and* $e_0 \xrightarrow{+}{}^{*} e_1$ *then* $\ell_0; \cdot \Vdash e_1$.

Proof Sketch. By a preservation argument.

Case: $(\mathit{unop}\,(\!(v_0)\!)^{\overline{\ell}_0})^{\ell_1} \rhd^{+} (\mathsf{TagErr})^{\ell_1}$
by definition $\ell_1; \cdot \Vdash (\mathsf{TagErr})^{\ell_1}$

Case: $(\mathit{unop}\,(\!(v_0)\!)^{\overline{\ell}_0})^{\ell_1} \rhd^{+} (\!(\delta(\mathit{unop}, v_0))\!)^{\overline{\ell}_0\ell_1}$

1. $\overline{\ell}_0$ is either all deep labels or a mix of shallow and untyped, by deep-label consistency of the redex
2. similarly, ℓ_1 must match $\overline{\ell}_0$
3. v_0 is a pair, because δ is defined on it
4. both components of v_0 are well-labeled, again by deep-label consistency on the redex
5. by the definition of δ

Case: $(\mathit{binop}\,(\!(v_0)\!)^{\overline{\ell}_0}\,(\!(v_1)\!)^{\overline{\ell}_1})^{\ell_2} \rhd^{+} (\mathsf{TagErr})^{\ell_2}$
by the definition of $\Vdash$

Case: $(\mathit{binop}\,(\!(v_0)\!)^{\overline{\ell}_0}\,(\!(v_1)\!)^{\overline{\ell}_1})^{\ell_2} \rhd^{+} (\delta(\mathit{binop}, v_0, v_1))^{\ell_2}$
by the definition of $\Vdash$ and δ (binary operators are not elimination forms in the model, thus the number computed by δ does not acquire labels from v_0 and v_1)

Case: $(\mathsf{app}\,(\!(v_0)\!)^{\overline{\ell}_0}\ v_1)^{\ell_1} \rhd^{+} (\mathsf{TagErr})^{\ell_1}$
by the definition of $\Vdash$

Case: $(\mathsf{app}\,(\!(\lambda x_0.\, e_0)\!)^{\overline{\ell}_0}\ v_0)^{\ell_1} \rhd^{+} (\!(e_0[x_0 \leftarrow (\!(v_0)\!)^{\ell_1 \mathit{rev}(\overline{\ell}_0)}])\!)^{\overline{\ell}_0\ell_1}$

1. $\overline{\ell}_0$ is all deep or a mix of shallow and untyped, by deep-label consistency of the redex
2. $\ell_1; \cdot \Vdash v_0$, also by deep-label consistency of the redex
3. let ℓ_n be the rightmost label in the sequence $\overline{\ell}_0$
4. $\ell_n; \cdot \Vdash (\!(v_0)\!)^{\ell_1 \mathit{rev}(\overline{\ell}_0)}$, by steps 1 and 2
5. $\ell_n; \cdot \Vdash x_0$ for each occurrence of x_0 in e_0, by deep-label consistency of the redex
6. by a substitution lemma

Case: $(\mathsf{app}\,(\!(\lambda(x_0 : \tau_0).\, e_0)\!)^{\overline{\ell}_0}\ v_0)^{\ell_1} \rhd^{+}$
$(\!(e_0[x_0 \leftarrow (\!(v_0)\!)^{\ell_1 \mathit{rev}(\overline{\ell}_0)}])\!)^{\overline{\ell}_0\ell_1}$
similar to the previous case

Case: $(\mathsf{app}\,(\!(\lambda(x_0 : \mathsf{scan}\, \sigma_0).\, e_0)\!)^{\overline{\ell}_0}\ v_0)^{\ell_1} \rhd^{+} (\mathsf{ScanErr})^{\ell_1}$
by the definition of $\Vdash$

Case: $(\mathsf{app}\,(\!(\lambda(x_0 : \mathsf{scan}\, \sigma_0).\, e_0)\!)^{\overline{\ell}_0}\ v_0)^{\ell_1} \rhd^{+}$
$(\!(e_0[x_0 \leftarrow (\!(v_0)\!)^{\ell_1 \mathit{rev}(\overline{\ell}_0)}])\!)^{\overline{\ell}_0\ell_1}$
similar to the other substitution cases

Case: $(\mathsf{app}\,(\!(\mathbb{G}\, \tau_0 \rightarrow \tau_1\ (v_0)^{\ell_0})\!)^{\overline{\ell}_1}\ v_1)^{\ell_2} \rhd^{+}$
$(\!(\mathsf{wrap}\ \tau_1\ (\mathsf{app}\ v_0\ (\mathsf{wrap}\ \tau_0\ (\!(v_1)\!)^{\ell_2 \mathit{rev}(\overline{\ell}_1)}))^{\ell_0}\,)\!)^{\overline{\ell}_1\ell_2}$

1. $\ell_0; \cdot \Vdash v_0$, by deep-label consistency of the redex
2. $\ell_2; \cdot \Vdash v_1$, again by deep-label consistency
3. $\overline{\ell}_1$ is either all deep or a mix of shallow and untyped, again by the consistency of the redex
4. by the definition of $\Vdash$

Case: $(\mathsf{noop}\ (\!(v_0)\!)^{\overline{\ell}_0})^{\ell_1} \rhd^{+} (\!(v_0)\!)^{\overline{\ell}_0\ell_1}$
by the definition of $\rightsquigarrow$, because a noop boundary connects either two deep components or a mix of shallow and untyped components (self edges or $\mathcal{S}$ to $\mathcal{U}$)

Case: $(\mathsf{scan}\, \sigma_0\ (\!(v_0)\!)^{\overline{\ell}_0})^{\ell_1} \rhd^{+} (\mathsf{ScanErr})^{\ell_1}$
by the definition of $\Vdash$

Case: $(\mathsf{scan}\, \sigma_0\ (\!(v_0)\!)^{\overline{\ell}_0})^{\ell_1} \rhd^{+} (\!(v_0)\!)^{\overline{\ell}_0\ell_1}$
by the definition of $\rightsquigarrow$, because a scan boundary links only shallow and/or untyped components

Case: $(\mathsf{wrap}\ \tau_0\ (\!(v_0)\!)^{\overline{\ell}_0})^{\ell_1} \rhd^{+} (\mathsf{WrapErr})^{\ell_1}$

by the definition of $\Vdash$

Case: $(\mathsf{wrap}\,(\tau_0 \rightarrow \tau_1)\,(\!(v_0)\!)^{\overline{\ell_0}})^{\ell_1} \rhd^{+} (\mathbb{G}\,\tau_0 \rightarrow \tau_1\,(\!(v_0)\!)^{\overline{\ell_0}})^{\ell_1}$
by the definition of $\Vdash$

Case: $(\mathsf{wrap}\,(\tau_0 \times \tau_1)\,(\!(\langle v_0, v_1\rangle)\!)^{\overline{\ell_0}})^{\ell_1} \rhd^{+}$
$(\langle \mathsf{wrap}\,\tau_0\,(\!(v_0)\!)^{\overline{\ell_0}}, \mathsf{wrap}\,\tau_1\,(\!(v_1)\!)^{\overline{\ell_0}}\rangle)^{\ell_1}$
by the definition of $\Vdash$; the step makes a new pair

Case: $(\mathsf{wrap}\,\tau_0\,(\!(v_0)\!)^{\overline{\ell_0}})^{\ell_1} \rhd^{+} (v_0)^{\ell_1}$
where $\tau_0 \in \mathsf{Int} \cup \mathsf{Nat}$ and *shape-match* (τ_0, v_0)
by the definition of $\Vdash$

□

Lemma A.2 (compilation). *If* $\vdash_s s_0 : T$ *then* $\vdash_s s_0 : T \rightsquigarrow e_0$ *and either:*

- $T \in \tau$ *and* $\vdash_{\mathcal{D}} e_0 : T$
- $T \in \lfloor\tau\rfloor$ *and* $\vdash_{\mathcal{S}} e_0 :$ **shape** (T)
- $T \in \mathcal{U}$ *and* $\vdash_{\mathcal{U}} e_0 : \mathcal{U}$

Lemma A.3 (type progress). *If* $\vdash_s e_0 : T$ *then either* $e_0 \in v \cup \mathsf{Error}$ *or* $e_0 \longrightarrow e_1$

Lemma A.4 (type preservation). *If* $\vdash_s e_0 : T$ *and* $e_0 \longrightarrow e_1$ *then* $\vdash_s e_1 : T$

Lemma A.5 (decomposition). *For all* e_0 *there exists unique* e_1, E_0 *such that* $e_0 = E_0[e_1]$

Lemma A.6. *If* $\vdash_{\mathcal{S}} e_0 : \sigma_0$ *then* $\vdash_{\mathcal{U}} e_0 : \mathcal{U}$

Proof. By definition, in particular because the untyped rules allow shape-annotated functions. □

Lemma A.7 (boundary-crossing).

- *If* $\vdash_{L} v_0 : T$ *and shape-match* (σ_0, v_0) *then* $\vdash_{\mathcal{S}} v_0 : \sigma_0$
- *If* $\vdash_{\mathcal{S}} v_0 : \sigma_0$ *then* $\vdash_{\mathcal{U}} v_0 : \mathcal{U}$
- *If* $\vdash_{\mathcal{D}} v_0 : \tau_0$ *and* $\mathsf{wrap}\,\tau_0\,v_0 \rhd v_1$ *then* $\vdash_{\mathcal{S}} v_1 :$ **shape** (τ_0) *and* $\vdash_{\mathcal{U}} v_1 : \mathcal{U}$

Lemma A.8 (owner preservation). *If* $\vdash e_0 : T$ *and* $\ell_0 \Vdash e_0$ *and* $e_0 \rhd e_1$ *then* $\ell_0 \Vdash e_1$

A.4 Expressiveness by Shallow: Example Programs

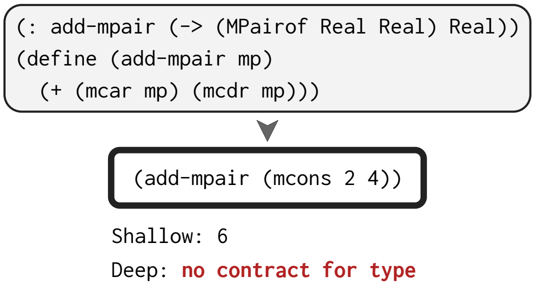
A.4.1 Less-strict Top Type.

Inspired by the following messages to the Racket-Users mailing list:

- *error : Attempted to use a higher-order value passed as 'Any' in untyped code*, sent by Denis Michiels on 2018-04-16. groups.google.com/g/racket-users/c/cCQ6dRNybDg/m/CKXgX1PyBgAJ accessed 2020-12-15
- *Typed Racket: 'Unable to protect opaque value passed as 'Any'' with interesting behavior*, sent by Marc Kaufmann on 2019-12-11. groups.google.com/g/racket-users/c/jtmVDFCGL28/m/jwl4hsjtBQAJ accessed 2020-12-15

Deep Racket enforces the top type with a wrapper that prevents clients from inspecting the enclosed value. This wrapper is a surprise for developers who expect programs such as figure 14 to run without error. This program defines a mutable box in typed code, assigns the top type (Any) to the box, and sends it to untyped code. The untyped module attempts to set the box. Deep Racket raises an exception when untyped code tries to modify the box. Unfortunately for the programmer, this error is essential for soundness. If untyped code put an integer in the box, then typed uses of the box would give a result that is inconsistent with its type. Shallow Racket runs the program without error because of its delayed checking strategy. If shallow-typed code tries to read a symbol from the box, that access will raise an error.

Other top types for higher-order values have similar behavior. For example, Shallow Racket can import a function at the general `Procedure` type, cast to a more specific type, and apply the function. A Deep cast only adds a second wrapper atop the restrictive wrapper for the `Procedure` type.



```
(: add-mpair (-> (MPairof Real Real) Real))
(define (add-mpair mp)
  (+ (mcar mp) (mcdr mp)))
```

```
(add-mpair (mcons 2 4))
```

```
Shallow: 6
Deep: no contract for type
```

Figure 24: Deep lacks wrappers for mutable pairs and a few other uncommon datatypes

A.4.2 No Missing Wrappers.

Several little-used types in Deep Racket lack wrappers (12 in total). Figure 24 demonstrates the issue with a mutable pair (`MPairof`) type. Deep Racket raises a run-time error when untyped code tries to call the `add-mpair` function. Shallow can run the program.

A.4.3 Uniform Behavior.

Inspired by the following messages to the Racket-Users mailing list:

- *Typed code from untyped code*, sent by Bertrand on 2020-02-17. groups.google.com/g/racket-users/c/UD20HadJ9Ec/m/Lmuw0U8mBwAJ accessed 2020-12-15
- *index-of + TR ... parametricity problem?*, sent by John B. Clements on 2019-12-15. groups.google.com/g/racket-users/c/ZbYRQCy93dY/m/kF_Ek0VvAQAJ accessed 2020-12-15

Figure 25 presents a typed module that imports an untyped function, `index-of`, with a precise polymorphic type. The wrapper that enforces this type creates a new wrapper for every input to the function—to enforce parametric polymorphism [33]. Unfortunately, these input wrappers change the behavior of `index-of`; it searches the list for a wrapped version of the symbol `'a` and returns a "not found" result (`#false`) instead of the correct position (`0`). Shallow Racket avoids all such changes in behavior because the transient semantics does not use wrappers to enforce types.

```
(index-of '(a b) 'a)
```

```
Untyped: 0
```

```
(require/typed racket/list
  [index-of
   (All (T)
    (-> (Listof T) T
        (U #false Natural)))])

(index-of '(a b) 'a)
```

```
Shallow: 0
Deep: #false
```

Figure 25: Deep contracts can change the behavior of code

References

[1] Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. 2014. Confined gradual typing. In *OOPSLA*, 251–270. DOI: 10.1145/2660193.2660222.

[2] Esteban Allende, Johan Fabry, and Éric Tanter. 2013. Cast insertion strategies for gradually-typed objects. In *DLS*, 27–36. DOI: 10.1145/2508168.2508171.

[3] Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. 2012. Typing the numeric tower. In *PADL*, 289–303. DOI: 10.1007/978-3-642-27694-1_21.

[4] Anonymous Author(s). 2022. Gradual soundness: lessons from Static Python. In *Submitted for publication*.

[5] Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. 2017. Sound gradual typing: only mostly dead. *PACMPL*, 1, OOPSLA, 54:1–54:24. DOI: 10.1145/3133878.

[6] Gavin Bierman, Martin Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP*, 257–281. DOI: 10.1007/978-3-662-44202-9_11.

[7] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. 2009. Thorn: robust, concurrent, extensible scripting on the JVM. In *OOPSLA*, 117–136. DOI: 10.1145/1640089.1640098.

[8] Giuseppe Castagna and Victor Lanvin. 2017. Gradual typing with union and intersection types. *PACMPL*, 1, ICFP, 41:1–41:28. DOI: 10.1145/3110285.

[9] Benjamin Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. 2018. KafKa: gradual typing for objects. In *ECOOP*, 12:1–12:23. DOI: 10.4230/LIPIcs.ECOOP.2018.12.

[10] Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. 2007. Advanced macrology and the implementation of Typed Scheme. In *SFP. Université Laval, DIUL-RT-0701*, 1–14. http://www2.ift.ulaval.ca/~dadub100/sfp2007/procPaper1.pdf.

[11] Christos Dimoulas, Robert Bruce Findler, and Matthias Felleisen. 2013. Option contracts. In *OOPSLA*, 475–494. DOI: 10.1145/2509136.2509548.

[12] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. 2011. Correct blame for contracts: no more scapegoating. In *POPL*, 215–226.

[13] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete monitors for behavioral contracts. In *ESOP*, 214–233. DOI: 10.1007/978-3-642-28869-2_11.

[14] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press. ISBN: 978-0-262-06275-6.

[15] Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. Collapsible contracts: fixing a pathology of gradual typing. *PACMPL*, 2, OOPSLA, 133:1–133:27. DOI: 10.1145/3276503.

[16] Robert Bruce Findler and Matthias Blume. 2006. Contracts as Pairs of Projections. Tech. rep. TR-2006-01. University of Chicago. DOI: 10.1007/11737414_16.

[17] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *ICFP*, 48–59. DOI: 10.1145/581478.581484.

[18] Matthew Flatt. 2016. Binding as sets of scopes. In *POPL*, 705–717. DOI: 10.1145/2837614.2837620.

[19] Matthew Flatt. 2013. Submodules in Racket: you want it *When*, again? In *GPSE*, 13–22. DOI: 10.1145/2517208.2517211.

[20] Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. 2012. Macros that work together: compile-time bindings, partial expansion, and definition contexts. *JFP*, 22, 2, 181–216. DOI: 10.1017/S0956796812000093.

[21] Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting gradual typing. In *POPL*, 429–442. DOI: 10.1145/2837614.2837670.

[22] Michael Greenberg. 2015. Space-efficient manifest contracts. In *POPL*, 181–194. DOI: 10.1145/2676726.2676967.

[23] Ben Greenman. 2020. *Deep and Shallow Types*. Ph.D. Dissertation. Northeastern University. ISBN: 9798557043649.

[24] [SW Rel.] Ben Greenman, Deep and Shallow Types for Gradual Languages (source code) version 1.0, 2022. VCS: https://github.com/bennn/g-pldi-2022, SWHID: ⟨swh:1:dir:2f1f76cafb72491d8526d18ae556499065ac6853⟩.

[25] [SW] Ben Greenman, Deep and Shallow Types for Gradual Languages v1.0 version 1.0, Apr. 2022. DOI: 10.5281/zenodo.6498926.

[26] Ben Greenman and Matthias Felleisen. 2018. A spectrum of type soundness and performance. *PACMPL*, 2, ICFP, 71:1–71:32. DOI: 10.1145/3236766.

[27] Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2019. Complete monitors for gradual types. *PACMPL*, 3, OOPSLA, 122:1–122:29. DOI: 10.1145/3360548.

[28] Ben Greenman, Lukas Lazarek, Christos Dimoulas, and Matthias Felleisen. 2022. A transient semantics for Typed Racket. *<Programming>*, 6, 2, 9:1–9:26. DOI: 10.22152/programming-journal.org/2022/6/9.

[29] Ben Greenman and Zeina Migeed. 2018. On the cost of type-tag soundness. In *PEPM*, 30–39. DOI: 10.1145/3162066.

[30] Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. 2019. How to evaluate the performance of gradual type systems. *JFP*, 29, e4, 1–45. DOI: 10.1017/S0956796818000217.

[31] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. 2006. Sage: hybrid checking for flexible specifications. In *SFP. University of Chicago, TR-2006-06*, 93–104. http://scheme2006.cs.uchicago.edu/scheme2006.pdf.

[32] [SW Rel.], GTP Benchmarks version 6.0, 2020. VCS: https://github.com/bennn/gtp-benchmarks, SWHID: ⟨swh:1:rev:d1a85945b64d8f49876f8921ef7a7bcdd82cc96f⟩.

[33] Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. 2007. Relationally-parametric polymorphic contracts. In *DLS*, 29–40. DOI: 10.1145/1297081.1297089.

[34] Fritz Henglein. 1994. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22, 3, 197–230. DOI: 10.1016/0167-6423(94)00004-2.

[35] David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *HOSC*, 23, 2, 167–189. DOI: 10.1007/s10990-011-9066-z.

[36] Khurram A. Jafery and Jana Dunfield. 2017. Sums of uncertainty: refinements go gradual. In *POPL*, 804–817. DOI: 10.1145/3009837.3009865.

[37] Matthias Keil and Peter Theimann. 2015. Blame assignment for higher-order contracts with intersection and union. In *ICFP*, 375–386. DOI: 10.1145/2784731.2784737.

[38] Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. Toward efficient gradual typing for structural types via coercions. In *PLDI*, 517–532. DOI: 10.1145/3314221.3314627.

[39] Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2021. How to evaluate blame for gradual types. *PACMPL*, 5, ICFP, 68:1–68:29. DOI: 10.1145/3473573.

[40] Jacob Matthews and Robert Bruce Findler. 2009. Operational semantics for multi-language programs. *TOPLAS*, 31, 3, 1–44. DOI: 10.1145/1498926.1498930.

[41] Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 3, 348–375. DOI: 10.1016/0022-0000(78)90014-4.

[42] Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. 2016. Extensible access control with authorization contracts. In *OOPSLA*, 214–233.

[43] Cameron Moy, Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2021. Corpse reviver: sound and efficient gradual typing via contract verification. *PACMPL*, 5, POPL, 1–28. DOI: 10.1145/3434334.

[44] Fabian Muehlboeck and Ross Tate. 2017. Sound gradual typing is nominally alive and well. *PACMPL*, 1, OOPSLA, 56:1–56:30. DOI: 10.1145/3133880.

[45] Fabian Muehlboeck and Ross Tate. 2021. Transitioning from structural to nominal code with efficient gradual typing. *PACMPL*, 5, OOPSLA, 127:1–127:29. DOI: 10.1145/3485504.

[46] Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2019. Size-change termination as a contract: dynamically and statically enforcing termination for higher-order programs. In *PLDI*, 845–859. DOI: 10.1145/3314221.3314643.

[47] Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2018. Soft contract verification for higher-order stateful programs. *PACMPL*, 2, POPL, 51:1–51:30. DOI: 10.1145/3158139.

[48] [SW Rel.], Racket version 7.8.0.5+ revision 7c9038, 2020. VCS: https://github.com/racket/racket, SWHID: ⟨swh:1:rev:7c903871bd8cb4bd32ed7188c180b5124f9bc201⟩.

[49] Gregor Richards, Ellen Arteca, and Alexi Turcotte. 2017. The vm already knew that: leveraging compile-time knowledge to optimize gradual typing. *PACMPL*, 1, OOPSLA, 55:1–55:27. DOI: 10.1145/3133879.

[50] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete types for TypeScript. In *ECOOP*, 76–100. DOI: 10.4230/LIPIcs.ECOOP.2015.76.

[51] Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2019. Transient typechecks are (almost) free. In *ECOOP*, 15:1–15:29. DOI: 10.4230/LIPIcs.ECOOP.2019.5.

[52] Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *SFP. University of Chicago, TR-2006-06*, 81–92. http://scheme2006.cs.uchicago.edu/scheme2006.pdf.

[53] Jeremy G. Siek, Peter Thiemann, and Philip Wadler. 2021. Blame and coercion: together again for the first time. *JFP*, 31, e20. DOI: 10.1017/S0956796821000101.

[54] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined criteria for gradual typing. In *SNAPL*, 274–293. DOI: 10.4230/LIPIcs.SNAPL.2015.274.

[55] Jeremy Siek, Peter Thiemann, and Philip Wadler. 2015. Blame and coercion: together again for the first time. In *PLDI*, 425–435. DOI: 10.1145/2737924.2737968.

[56] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. Chaperones and impersonators: run-time support for reasonable interposition. In *OOPSLA*, 943–962. DOI: 10.1145/2384616.2384685.

[57] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is sound gradual typing dead? In *POPL*, 456–468. DOI: 10.1145/2837614.2837630.

[58] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual typing for first-class classes. In *OOPSLA*, 793–810. DOI: 10.1145/2384616.2384674.

[59] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as libraries. In *PLDI*, 132–141. DOI: 10.1145/1993498.1993514.

[60] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage migration: from scripts to programs. In *DLS*, 964–974. DOI: 10.1145/1176617.1176755.

[61] Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical types for untyped languages. In *ICFP*, 117–128. DOI: 10.1145/1863543.1863561.

[62] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of Typed Scheme. In *POPL*, 395–406. DOI: 10.1145/1328438.1328486.

[63] Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. 2017. Migratory typing: ten years later. In *SNAPL*, 17:1–17:17. DOI: 10.4230/LIPIcs.SNAPL.2017.17.

[64] Matías Toro and Éric Tanter. 2017. A gradual interpretation of union types. In *SAS*, 382–404. DOI: 10.1007/978-3-319-66706-5_19.

[65] Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. 2018. The behavior of gradual types: a user study. In *DLS*, 1–12. DOI: 10.1145/3276945.3276947.

[66] [SW Rel.], Typed Racket version 1.12+ revision c074c93, 2020. VCS: https://github.com/bennn/typed-racket, SWHID: ⟨swh:1:rev:c074c9333e467cb7cd2058511ac63a1d51b4948e⟩.

[67] Tom Van Cutsem and Mark S Miller. 2013. Trustworthy proxies. In *ECOOP*, 154–178. DOI: 10.1007/978-3-642-39038-8_7.

[68] Michael M. Vitousek, Andrew Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and evaluation of gradual typing for Python. In *DLS*, 45–56. DOI: 10.1145/2661088.2661101.

[69] Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. 2019. Optimizing and evaluating transient gradual typing. In *DLS*, 28–41. DOI: 10.1145/3359619.3359742.

[70] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big types in little runtime: open-world soundness and collaborative blame for gradual type systems. In *POPL*, 762–774. DOI: 10.1145/3009837.3009849.

[71] Andrew K. Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and Computation*, 38–94. First appeared as Technical Report TR160, Rice University, 1991. DOI: 10.1006/inco.1994.1093.

[72] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. Integrating typed and untyped code in a scripting language. In *POPL*, 377–388. DOI: 10.1145/1706299.1706343.