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

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 meta-theorems: 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
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.
- 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]. Natural interprets types as contracts in a straightforward manner. 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].

1Natural is a.k.a. Guarded [68], Behavioral [9], and Deep [65].
2Researchers 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.
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 $\texttt{txt}$ 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 \texttt{wrap} term applies a contract, a \texttt{scan} term performs a first-order (predicate) check, and a \texttt{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 (\texttt{app s s}), primitive operation applications (\texttt{unop s}, \texttt{binop s s}), variables $x$, integers $i$, pairs $(s, s)$, 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 ($\lambda(x : \tau). s$). The underline is a syntax hint that only the top-level shape of this type is guaranteed at run-time. Types $\tau$ express natural numbers (Nat), integers (Int), pairs ($\tau \times \tau$), and functions ($\tau \rightarrow \tau$). Modules associate a label with an expression (module $L s$). The label $L$ is either $D$ for deep-typed code, $S$ for shallow-typed code, or $U$ for untyped code. For example, the term (module $D s_0$) says that $s_0$ is a deep-typed expression. Any module expressions within $s_0$ are free to use any typing style ($D, S,$ or $U$).

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 $\Gamma$ and an expression $s$ to a result specification. A result $T$ is either a type $\tau$ for deep-typed code, an underlined type $\underline{\tau}$ for shallow-typed code, or the uni-type $\ulcorner U \urcorner$ 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 ($\Lambda$). 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 ($\sigma$) 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 ($\mathcal{G} \langle \tau_0 \rightarrow \tau_1 \rangle v_0$) provides type-restricted access to a function. Shallow-typed functions have
a shape annotation and a scan tag in the evaluation syntax ($\lambda(x : \text{scan}\,\sigma), e$) 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).

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

![Figure 5: Evaluation syntax](image)

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 ($\Gamma \vdash_T$) validates full types, the shallow judgment ($\Gamma \vdash_S$) checks top-level type shapes, and the untyped judgment ($\Gamma \vdash_U$) 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 (fst x0) 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.
Applications in deep (and untyped) code recur on their subexpressions. Applications in shallow code insert an additional `let` to their argument to indicate the need for a domain check, because untyped code can potentially invoke these functions. 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:

\[
\Gamma \vdash x : \sigma
\]
\[
\begin{align*}
(x_0 : \sigma_0) &\in \Gamma \\
\Gamma \vdash x_0 : \sigma_0
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash e_0 : \sigma_0 \\
\Gamma \vdash \lambda x_0. e_0 : \sigma_0
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash \lambda (x_0 : \sigma_0) . e_0 : \sigma_0 \\
\Gamma \vdash \lambda (x_0 : \sigma_0) . e_0 : \sigma_0
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash e_0 : \sigma_0 \\
\Gamma \vdash \text{app} e_0 : \text{Top}
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash \text{noop} e_0 : \text{Top}
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash \text{scan} e_0 : \sigma_0 \\
\Gamma \vdash \text{wrap} e_0 : \sigma_0
\end{align*}
\]

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

\[
\Gamma \vdash e : \mathcal{U}
\]
\[
\begin{align*}
(x_0 : \mathcal{U}) &\in \Gamma \\
\Gamma \vdash x_0 : \mathcal{U}
\end{align*}
\]
\[
\begin{align*}
(x_0 : \mathcal{U}), \Gamma \vdash e_0 : \mathcal{U}
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash \lambda x_0. e_0 : \mathcal{U}
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash e_0 : \mathcal{U}
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash \text{app} e_0 : \mathcal{U}
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash \text{noop} e_0 : \mathcal{U}
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash \text{scan} e_0 : \mathcal{U}
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash \text{wrap} e_0 : \mathcal{U}
\end{align*}
\]

**Figure 8:** Untyped typing judgment (selected rules)
\[
\begin{align*}
\text{unop } v_0 & \Rightarrow \text{TagErr} \\
& \quad \text{if } \delta(\text{unop}, v_0) \text{ is undefined} \\
\text{unop } v_0 & \Rightarrow \delta(\text{unop}, v_0) \\
& \quad \text{if } \delta(\text{unop}, v_0) \text{ is defined} \\
\text{binop } v_0 v_1 & \Rightarrow \text{TagErr} \\
& \quad \text{if } \delta(\text{binop}, v_0, v_1) \text{ is undefined} \\
\text{binop } v_0 v_1 & \Rightarrow \delta(\text{binop}, v_0, v_1) \\
& \quad \text{if } \delta(\text{binop}, v_0, v_1) \text{ is defined} \\
\app v_0 v_1 & \Rightarrow \text{TagErr} \\
& \quad \text{if } \neg \text{shape-match} (\text{Fun}, v_0) \\
\app (\lambda x_0. e_0) v_0 & \Rightarrow e_0[x_0 \leftarrow v_0] \\
\app (\lambda (x_0 : \tau_0). e_0) v_0 & \Rightarrow e_0[x_0 \leftarrow v_0] \\
\app (\lambda (x_0 : \text{scan } \sigma_0). e_0) v_0 & \Rightarrow \text{ScanErr} \\
& \quad \text{if } \neg \text{shape-match} (\sigma_0, v_0) \\
\app (\lambda (x_0 : \text{scan } \sigma_0). e_0) v_0 & \Rightarrow e_0[x_0 \leftarrow v_0] \\
& \quad \text{if } \text{shape-match} (\sigma_0, v_0) \\
\app (\text{\textsf{G}} (\tau_0 \rightarrow \tau_1) v_0) v_1 & \Rightarrow \text{wrap } \tau_1 (\app v_0 (\text{wrap } \tau_0 v_1)) \\
\text{noop } v_0 & \Rightarrow v_0 \\
\text{scan } \sigma_0 v_0 & \Rightarrow \text{ScanErr} \\
& \quad \text{if } \neg \text{shape-match} (\sigma_0, v_0) \\
\text{scan } \sigma_0 v_0 & \Rightarrow v_0 \\
& \quad \text{if } \text{shape-match} (\sigma_0, v_0) \\
\text{wrap } \tau_0 v_0 & \Rightarrow \text{WrapErr} \\
& \quad \text{if } \neg \text{shape-match} (\text{shape } (\tau_0), v_0) \\
\text{wrap } (\tau_0 \rightarrow \tau_1) v_0 & \Rightarrow \text{\textsf{G}} (\tau_0 \rightarrow \tau_1) v_0 \\
& \quad \text{if } \text{shape-match} (\text{Fun}, v_0) \\
\text{wrap } (\tau_0 \times \tau_1) (v_0, v_1) & \Rightarrow (\text{\textsf{wrap } \tau_0 v_0, \text{\textsf{wrap } \tau_1 v_1}) \\
\text{wrap } \tau_0 v_0 & \Rightarrow v_0 \\
& \quad \text{if } \tau_0 \in \text{Int } \cup \text{Nat } \text{and } \text{shape-match} (\tau_0, v_0)
\end{align*}
\]

\[
\begin{align*}
\text{e} \Rightarrow^* \text{e} & \overset{\text{def}}{=} \text{reflexive, transitive, compatible} \\
& \quad \text{(w.r.t. } E \text{) closure of } \Rightarrow \\
\text{s} \Rightarrow^* \text{e} & \overset{\text{def}}{=} \exists T, e_1. \\
& \quad \tau_s : T \xrightarrow{e_1} e_1 \land e_1 \Rightarrow^* e
\end{align*}
\]

Figure 10: Semantics for the evaluation syntax (left) and a labeled variant (right)
Deep and Shallow Types for Gradual Languages

The right half of figure 10 presents a labeled notion of reduction for the evaluation language. By design, the reduction rules are identical to the basic rules from figure 10 except for superset 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 (\(\text{app} (\lambda x_0. e_0) v_0\)) 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 \(\text{rev}\) (figure 11) keeps these labels in proper order by reversing them—because the argument value flows into 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 \(\ell_1\) 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 \(\langle v \rangle^\ell\) lets any value appear under an arbitrary number of labels. These labels correspond to modules from the surface syntax, and thus combine a kind \(\langle D, S, \ell U \rangle\) 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: \((\text{wrap} \tau_0 v_0)\), \((\text{scan} \sigma_0 v_0)\), \((\text{noop} v_0)\), and \((\text{G} \tau_0 v_0)\).
- To reduce parenthesis and superscripts, the abbreviation \((\ell_0)\) captures a sequence of labels. For example:

\[ e = \langle x \rangle | v | \langle e, e \rangle | \text{unop} e | \text{binop} e e | \text{app} e e | \text{Error} | \text{wrap} \tau (e)^\ell | \text{scan} \sigma (e)^\ell | \text{noop} (e)^\ell | (e)^\ell \]

\[ v = i | (v, v') | \lambda x. e | \lambda (x: \text{scan} \sigma). e | \text{G} (\tau \rightarrow \tau) (v)^\ell | (v)^\ell \]

\[ E = \ldots | (E)^\ell \]

\[ \ell = D_0 | D_1 | \ldots | S_0 | S_1 | \ldots | V_0 | V_1 | \ldots \]

\[ \ell_0 = \cdot | (x: \ell), \ell \]

Abbreviation: \((\ell_0)\ldots(\ell_n)^\ell = \langle \ell_0 \rangle^{\ell_0} \ldots^{\ell_n} \]

**Figure 12:** Labeled evaluation syntax

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.
Type soundness predicts the possible outcomes of a well-typed redex. The judgment allows any mix of shallow ($S$) and untyped ($U$) labels around an expression, but restricts the use of deep labels ($D$). Concretely, the judgment analyzes an expression relative to a context label and an environment ($L$). 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 must have only deep labels around it. 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 $k$ and a characterization function $F$ that maps a subset of the surface types $T$ to an evaluation-language type (either $\tau$, $\sigma$, or $U$).

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

- $s_0 \rightarrow^* v_0$ and $r_L v_0 : F (T)$
- $s_0 \rightarrow^* \text{Error}$
- $s_0 \rightarrow^* \text{diverges}$

There are three important characterization functions $F$ for the analysis: $1$ is the identity function on types $r$; $\text{shape}$ maps underlined types $\hat{r}$ to shapes $\sigma$ (similar to $\text{shape}$ from figure 7); and $0$ maps $U$ to $U$.

**Theorem 3.2 (Type soundness).**

- Language $D$ satisfies $\text{TS ($r_D$, 1)}$
- Language $S$ satisfies $\text{TS ($r_S$, shape)}$
- Language $U$ satisfies $\text{TS ($r_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 $r_D e_0 : \tau_0$ or $\tau_\neq e_0 : \sigma_0$) and $e_0 \not\rightarrow e_1$, then $e_1 \not\in \text{TagErr}$.

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 $r$ judgment (figure 13).

**Theorem 3.4 (Complete monitoring).** If $r_s s_0 : T \rightarrow e_0$ and $\ell_0 ; \ell_1 \vdash e_0$ and $e_0 \rightarrow^{\ast} e_1$ then $\ell_0 ; \ell_1 \vdash e_1$.

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

**Case:** (app $\langle \lambda x_0. e_0 \rangle \tilde{\ell}_0 \ell_0 \ell_1 (\langle x_0 \rightarrow \langle v_0 \rightarrow (\text{rev}(\ell_0)) \rangle \tilde{\ell}_0 \ell_1 \ell_2$)

1. $\ell_0$ is all deep or a mix of shallow and untyped, by deep-label consistency of the redex
2. $\ell_1 ; \ell_2 \vdash v_0$, also by deep-label consistency of the redex
3. let $\ell_n$ be the rightmost label in the sequence $\ell_0 \ell_1 \ell_2$
4. $\ell_n ; \ell_2 \vdash (\langle v_0 \rightarrow (\text{rev}(\ell_0)) \rangle \ell_0 \ell_1 \ell_2$, by steps 1 and 2
5. $\ell_n ; \ell_1 \vdash x_0$ for each occurrence of $x_0$ in $e_0$, by deep-label consistency of the redex
6. by a substitution lemma

**Case:** (app $\langle \text{wrap} (\lambda \ell_1 (\langle v_0 \rightarrow (\text{rev}(\ell_0)) \rangle \ell_0 \ell_1 \ell_2$)

1. $\ell_0 ; \ell_1 \vdash v_0$, by deep-label consistency of the redex
2. $\ell_2 ; \ell_1 \vdash v_1$, again by deep-label consistency
3. $\ell_1$ is either all deep or a mix of shallow and untyped, both again by the consistency of the redex
4. by the definition of $\vdash$

**Case:** (noop $\langle \langle v_0 \rightarrow (\text{rev}(\ell_0)) \rangle \ell_0 \ell_1 \ell_2$

by the definition of $\rightarrow$, because a noop boundary connects either two deep components or a mix of shallow and untyped components (self edges or $S$ to $U$)

**Case:** (scan $\langle v_0 \rightarrow (\text{rev}(\ell_0)) \rangle \ell_0 \ell_1 \ell_2$

by the definition of $\rightarrow^* $, the ScanErr judgment (figure 13)
We have implemented three-way interactions atop Typed wrappers appear in a sibling submodule to delay the cost of was how to modify these compiler back-ends to produce The wrappers do not have types due to the organization of the exporting shallow module or the importing deep module. This raises type-annotated import statement and to generate contracts shallow-typed modules. Because untyped imports lack types, the contract. Switching a module from Deep to Shallow may require casts to meet the subtype. 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.

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.

Because deep and shallow Racket cannot share macros. 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 either untyped code or that generates a contract: (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.

### 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.

### 4.1 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, the enforcement of deep types, and metaprogramming.

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 type boundaries (section 4.4).

### 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 either untyped code or that generates a contract: (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.

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.
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 type-set in bold if they are the best (lowest) in their row.

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 deliverable if all of its configurations run at most $D$ times slower than the untyped configuration.

Figure 17 counts the proportion of $D$-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

Benchmark Best w/ D+S Benchmark Best w/ D+S

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Best w/ D+S</th>
<th>Benchmark</th>
<th>Best w/ D+S</th>
</tr>
</thead>
<tbody>
<tr>
<td>forth</td>
<td>12%</td>
<td>zordoz</td>
<td>47%</td>
</tr>
<tr>
<td>fsm</td>
<td>38%</td>
<td>lnm</td>
<td>66%</td>
</tr>
<tr>
<td>fsmoo</td>
<td>31%</td>
<td>suffixtree</td>
<td>48%</td>
</tr>
<tr>
<td>mbta</td>
<td>19%</td>
<td>snake</td>
<td>46%</td>
</tr>
<tr>
<td>morsecode</td>
<td>25%</td>
<td>take5</td>
<td>36%</td>
</tr>
<tr>
<td>zombie</td>
<td>6%</td>
<td>acquire</td>
<td>64%</td>
</tr>
<tr>
<td>dungeon</td>
<td>31%</td>
<td>tetris</td>
<td>46%</td>
</tr>
<tr>
<td>jpeg</td>
<td>38%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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. The author of this library reported poor performance due to deep type boundaries. Changing a bridge module from deep to shallow 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 type-set in bold if they are the best (lowest) in their row.

**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.

5.3.3 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 type-set in bold if they are the best (lowest) in their row.

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

\[ Γ ⊢ T \]

\[ (x₀ : U) ∈ Γ \]

\[ Γ ⊢ x₀ : U \]

\[ Γ ⊢ x₀ : τ₀ \]

\[ Γ ⊢ x₀ : τ₀ \]

\[ Γ ⊢ x₀ : τ₀ \]

\[ Γ ⊢ x₀ : τ₀ \]

\[ Γ ⊢ n₀ : Nat \]

\[ Γ ⊢ n₀ : Nat \]

\[ Γ ⊢ s₀ : U \]

\[ Γ ⊢ s₁ : U \]

\[ Γ ⊢ s₀ : U \]

\[ Γ ⊢ s₁ : U \]

\[ Γ ⊢ s₀ : τ₀ \]

\[ Γ ⊢ s₁ : τ₁ \]

\[ △(unop, τ₀) = τ₁ \]

\[ △(unop, τ₀) = τ₁ \]

\[ △(unop, τ₀) = τ₁ \]

\[ △(binop, τ₀, τ₁) = τ₂ \]

\[ △(binop, τ₀, τ₁) = τ₂ \]

\[ △(binop, τ₀, τ₁) = τ₂ \]

\[ △(binop, τ₀, τ₁) = τ₂ \]

\[ △(binop, τ₀, τ₁) = τ₂ \]

\[ L₀ \]

\[ T₀ \]

\[ T₁ \]

\[ D \]

\[ D \]

\[ D \]

\[ S \]

\[ S \]

\[ S \]

\[ S \]

\[ U \]

\[ U \]

\[ T₁ \]

Figure 18: Surface 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 \( \ell \) names an source-code component. Expressions and values come with at least one ownership label; for example, \((42, 0)\) is an integer with one owner and \((((42, 0), 1)\) is an integer with three owners, written \(42, 0, 1\) for short. A complete monitoring theorem requires two ingredients that manage these labels. First, a reduction relation \( \rightarrow^* \) 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 20) 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:

\[
L_0, T_0 \vdash e_0 \\
S_0, T_0 \vdash e_0 \\
D_0, T_0 \vdash e_0
\]

Values such as \(42, 0, 1\) represent a communication that slipped past the run-time checking protocol, and therefore

Figure 20: Deep 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 \( r \) relation over the surface language. Recall that \( s t a t \) and \( d y n \) are boundary terms; they link two components, a context and an enclosed expression, via a type.

When reading an example, accept the transitions \( e \rightarrow r \) 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.
   \[
   \text{stat} \ (e_0 \cdot \text{Nat} \cdot e_1) \Downarrow \text{Nat} \ (0)^{e_0} \cdot 0^{e_1} \\
   \text{The value 0 fully matches the type Nat.}
   \]

2. Any other value that crosses a boundary must acquire the label of the new context.
   \[
   \text{stat} \ (e_0 \cdot \text{Nat} \cdot e_1) \Downarrow ((-2, 1))^{e_0} \cdot (-2, 1)^{e_1} \\
   \text{The pair } (-2, 1) \text{ does not match the type Nat.}
   \]

3. Every value that flows out of a value \( v_0 \) acquires the labels of \( v_0 \) and the context.
   \[
   \text{snd} \Downarrow (((1)^{e_0}, (2))^{e_1})^{e_0} \cdot (2)^{e_1} \\
   \text{The value 2 flows out of the pair } (1, 2).
   \]

4. Every value that flows into a function \( v_0 \) acquires the label of the context and the reversed labels of \( v_0 \).
   \[
   \text{app} \Downarrow ((\lambda x_0. \text{fst} x_0)^{e_0} \cdot (8, 6))^{e_1} \cdot (8, 6)^{e_1} \\
   \text{The argument value } (8, 6) \text{ is input to the function.}
   \]
Although guideline 4 talks about functions, it generalizes to complete monitoring.

### A.3 Lemmas for the Model

**Theorem A.1** (complete monitoring). If \( \tau s so : T \rightsquigarrow e_0 \) and \( \ell_0: \vdash e_0 \) and \( e_0 \xrightarrow{\tau} e_1 \) then \( \ell_0: \vdash e_1 \).

**Proof Sketch.** By a preservation argument.

**Case:** \( \text{unop}(\langle v_0 \rangle)^f \xrightarrow{\tau} (\text{TagErr})^{f_1} \)

by definition \( \ell_1: \vdash \text{TagErr} \)

**Case:** \( \text{unop}(\langle v_0 \rangle)^f \xrightarrow{\tau} (\delta(\text{unop}, v_0))^{f_1} \)

1. \( \bar{T}_0 \) is either all deep labels or a mix of shallow and untyped, by deep-label consistency of the redex
2. similarly, \( \ell_1 \) must match \( \bar{T}_0 \)
3. \( v_0 \) is a pair, because \( \delta \) 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 \( \delta \)

**Case:** \( \text{binop}(\langle v_0 \rangle)^f \langle (v_1) \rangle^{f_1} \xrightarrow{\tau} (\text{TagErr})^{f_2} \)

by the definition of \( \tau \)

**Case:** \( \text{binop}(\langle v_0 \rangle)^f \langle (v_1) \rangle^{f_1} \xrightarrow{\tau} (\delta(\text{binop}, v_0, v_1))^{f_2} \)

by the definition of \( \tau \) and \( \delta \) (binary operators are not elimination forms in the model, thus the number computed by \( \delta \) does not acquire labels from \( v_0 \) and \( v_1 \))

**Case:** \( \text{app}(\langle v_0 \rangle)^f \langle (v_1) \rangle^{f_1} \xrightarrow{\tau} (\text{TagErr})^{f_1} \)

by the definition of \( \tau \)

**Case:** \( \langle (v_0) \rangle^{f_1} \xrightarrow{\tau} (\langle (v_0) \rangle^{f_1}(\text{rev}(\bar{T}_0))) \bar{T}_1 \)

1. \( \bar{T}_0 \) is all deep or a mix of shallow and untyped, by deep-label consistency of the redex
2. \( \ell_1: \vdash v_0 \), also by deep-label consistency of the redex
3. let \( \ell_n \) be the rightmost label in the sequence \( \bar{T}_0 \)
4. \( \ell_n: \vdash (\langle (v_0) \rangle)^{f_1}(\text{rev}(\bar{T}_0)) \) by steps 1 and 2
5. \( \ell_n: \vdash x_0 \) for each occurrence of \( x_0 \) in \( e_0 \), by deep-label consistency of the redex
6. by a substitution lemma

**Case:** \( \langle (\lambda x_0: v_0). e_0 \rangle^{f_1} \xrightarrow{\tau} (\langle (\lambda x_0: v_0). e_0 \rangle^{f_1}(\text{rev}(\bar{T}_0)) \bar{T}_1 \)

similar to the previous case

**Case:** \( \langle (\lambda x_0: \text{scan} \sigma_0). e_0 \rangle^{f_1} v_0 \xrightarrow{\tau} (\text{ScanErr})^{f_1} \)

by the definition of \( \tau \)

**Case:** \( \langle (\lambda x_0: \text{scan} \sigma_0). e_0 \rangle^{f_1} v_0 \xrightarrow{\tau} (\langle (\lambda x_0: \text{scan} \sigma_0). e_0 \rangle^{f_1}(\text{rev}(\bar{T}_0)) \bar{T}_1 \)

similar to the other substitution cases

**Case:** \( \langle (\text{wrap}(\tau_1)(\langle (v_0) \rangle^{f_1}(\text{wrap}(\tau_1)(\langle (v_0) \rangle^{f_1}(\text{rev}(\bar{T}_0))))) \rangle^{f_1} \xrightarrow{\tau} \langle (\lambda x_0: (\langle (v_0) \rangle^{f_1}(\text{rev}(\bar{T}_0)))) \rangle \bar{T}_1 \)

1. \( \ell_0: \vdash v_0 \), by deep-label consistency of the redex
2. \( \ell_2: \vdash v_1 \), again by deep-label consistency
3. \( \bar{T}_1 \) is either all deep or a mix of shallow and untyped, again by the consistency of the redex
4. by the definition of \( \tau \)

**Case:** \( \langle (\text{noop}(\langle (v_0) \rangle)^f) v_0 \rangle^{f_1} \xrightarrow{\tau} (\text{WrapErr})^{f_1} \)

by the definition of \( \tau \) and \( \delta \), because a scan boundary links only shallow and/or untyped components

**Case:** \( \langle (\text{scan} \sigma_0 (\langle (v_0) \rangle)^f) \rangle^{f_1} \xrightarrow{\tau} (\text{ScanErr})^{f_1} \)

by the definition of \( \tau \)

**Case:** \( \langle (\text{scan} \sigma_0 (\langle (v_0) \rangle)^f) \rangle^{f_1} \xrightarrow{\tau} (\langle (v_0) \rangle)^{f_1} \bar{T}_1 \)

by the definition of \( \tau \), because a scan boundary links only shallow and/or untyped components
by the definition of $\vdash$

Case: $(\text{wrap } t_0 \rightarrow t_1) \leftarrow (v_0 \leftarrow v_1)\leftarrow \tilde{f}_1$
by the definition of $\vdash$

Case: $(\text{wrap } (t_0 \times t_1) \leftarrow (v_0, v_1)\leftarrow \tilde{f}_1$
by the definition of $\vdash$; the step makes a new pair

Case: $(\text{wrap } t_0 \leftarrow (v_0)\leftarrow \tilde{f}_1$
where $\tau \in \text{Int} \cup \text{Nat}$ and $\text{shape-match}(t_0, v_0)$
by the definition of $\vdash$

□

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

- $T \in T$ and $\vdash_o e_0 : T$
- $T \in \mathcal{L}$ and $\vdash_o e_0 : \mathcal{L}$
- $T \in \mathcal{E} \mathit{and} \vdash_s e_0 : \mathcal{E}$

Lemma A.3 (type progress). If $\vdash_s e_0 : T$ then either $e_0 \in \mathcal{V} \cup \mathit{Error}$ or $e_0 \rightarrow e_1$

Lemma A.4 (type preservation). If $\vdash_r e_0 : T$ and $e_0 \rightarrow e_1$ then $\vdash_r e_1 : T$

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 $\vdash_r e_0 : \sigma_0$ then $\vdash_r e_0 : \mathcal{U}$

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 $\text{shape-match}(\sigma_0, v_0)$ then $\vdash_S v_0 : \sigma_0$
- If $\vdash_r v_0 : \sigma_0$ then $\vdash_S v_0 : \mathcal{U}$
- If $\vdash_D v_0 : T_0$ and $\text{wrap } t_0 \leftarrow v_0 \Rightarrow v_1$ then $\vdash_S v_1 : \text{shape}(t_0)$ and $\vdash_U v_1 : \mathcal{U}$

Lemma A.8 (owner preservation). If $\vdash e_0 : T$ and $\ell_0 \vdash e_0$ and $e_0 \Rightarrow e_1$ 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/CKXgXtPYbGjAaccessed 2021-12-15
- 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/ZbYRQCq93dY/m/kF_EkADVAQAJ accessed 2020-12-15

Deep Racket forces 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.

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 Real) 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/UD28HadjEc/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/ZbYRQCq93dY/m/kF_EkADVAQAJ 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.
Figure 25: Deep contracts can change the behavior of code

References


