Honest and Lying Types Ben Greenman
Thesis Proposal October 1, 2019

Abstract

The literature on migratory typing presents many strategies for mixing typed and untyped code. Two strategies, *natural* and *transient*, are especially interesting because they guarantee type soundness and do not limit the expressiveness of untyped code. Despite these commonalities, however, the strategies offer vastly different guarantees and performance tradeoffs. Their complementary strengths suggest the need for a combination.

My work to date has developed novel methods for understanding the semantics and performance of migratory typing systems. I propose to leverage this expertise to: (1) design a semantics that supports interoperability between natural and transient, (2) implement the semantics, and (3) systematically evaluate the performance of the combination.

1 Migratory Typing: Theory vs. Practice

A migratory typing system adds static types to an existing dynamically-typed host language [30]. At a minimum, the addition requires a static type checker and syntax to accomodate mixed-typed code. If the types are intended as claims about the kinds of values that flow through a program at run-time, then the addition also requires a method of enforcing types.

Typed Racket [29] is one example of a migratory typing system. The language accepts type-annotated Racket programs, validates the annotations with a type checker, and enforces the annotations with a translation of types to higher-order contracts. For example, figure 1 presents a mixed-typed program consisting of three modules. The two untyped modules at the top of the figure define a guessing game and a game player. The typed module at the bottom gives the player five chances to submit a correct guess.

The typed driver module assigns a static type to the untyped game and game player. At compile-time, the type checker validates the contents of the driver module assuming that the types assigned to these untyped functions are correct. At run-time, higher-order contracts dynamically enforce the claims of the types. Thanks to the contracts, these types are kept honest. If a different player function were to submit a string as a guess, a contract violation would halt the program and direct the programmer to the boundary between the typed driver and the untyped player.

```
#lang racket
(provide play)

(define (play)
  (define n (random 10))
  (lambda (guess)
    (= guess n)))
```

```
#lang racket
(provide stubborn-player)

(define (stubborn-player i)
  4)
```

```
#lang typed/racket

(require/typed "guess-game.rkt"
  [play (-> (Natural -> Boolean))])
(require/typed "stubborn-player.rkt"
  [stubborn-player (Natural -> Natural)])

(define check-guess (play))

(for/or ([i : Natural (in-range 5)])
  (check-guess (stubborn-player i)))
```

Fig. 1: A mixed-typed Typed Racket program [15]

30 In theory, Typed Racket gives programmers the ability to freely mix typed and untyped modules. Imagine
31 a large, untyped codebase; its maintainers may add types to any single module while leaving the rest untyped
32 to arrive at a new, runnable program [28, 29]. After the conversion, the new module benefits from static
33 type checking, which enables type-driven compiler optimizations.
34 In practice, a programmer's freedom to add types is severly limited by the run-time cost of type enforce-
35 ment [27, 15]. Adding types to one module adds a contract boundary to its neighbors. These boundaries
36 may add overhead throughout the program. When two modules communicate through a boundary, they
37 may experience three kinds of performance overhead. First, there is the overhead of checking every value
38 that crosses the boundary. Second, a higher-order boundary must allocate new wrappers to constrain the
39 future behavior of any values that cross it. Third, wrapped values suffer from a layer of indirection. These
40 overheads can dramatically increase the running time of a program (figure 2). Clearly, keeping types honest
41 may impose a huge cost.

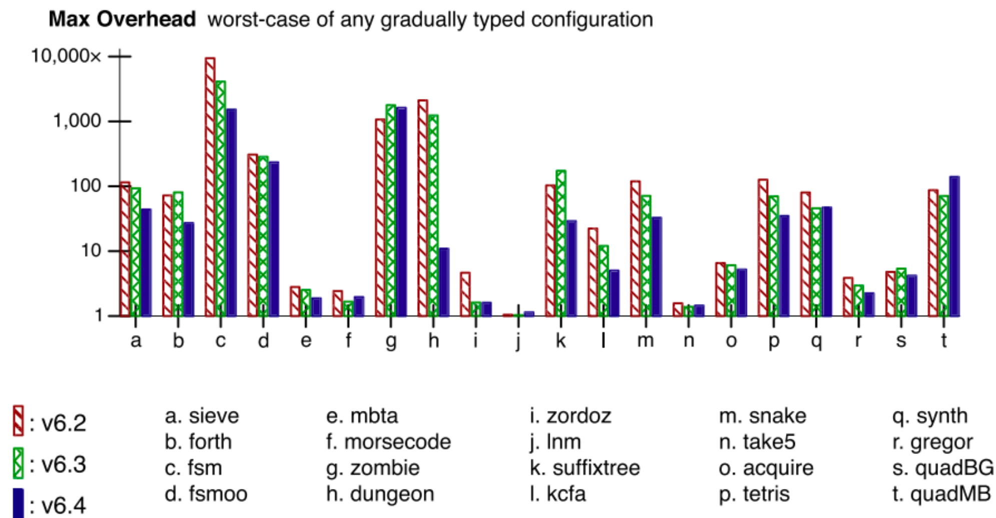

Fig. 2: Worst-case overheads across 20 benchmarks and 3 versions of Typed Racket [15]

42 ### 1.1 Different Strategies

43 Other migratory typing systems do not keep types honest in the same manner as Typed Racket. Some add a
44 runtime invariant to reduce the cost of honest types [34, 4, 19, 22]. Still others choose to selectively enforce
45 types; a value may be obliged to satisfy a type in some contexts, but not all [33]. With a few exceptions,[1]
46 the different strategies fall into four broad categories: natural, concrete, erasure, and transient.
47 Typed Racket implements the *natural* type enforcement strategy [18, 29]. A natural semantics carefully
48 guards the boundaries between typed and untyped code by wrapping higher-order values in proxies and by
49 eagerly checking/traversing other data. For example, if a natural language expects a list of numbers, then it
50 checks every element of an incoming list. If a natural language expects a function, it creates a proxy around
51 an incoming function value to protect future inputs and validate future results.
52 A *concrete* system comes with two invariants. First, only statically-typed code can create new values.
53 Second, every value has an immutable and precise type label. If a run-time system enforces these invariants,
54 then types can be kept honest through inexpensive label checks [19, 34, 4, 22].[2] For example, if a typed
55 function expects a vector of integers and receives a value from a dynamically-typed context, the function
56 can check whether the value's label is a subtype of the expected type. This operation is much simpler than
57 traversing and wrapping the vector.

[1]Like types let a programmer toggle between concrete types and erased types [34, 23]. Grace enforces user-supplied type annotations with tag checks [24]. Pyret enforces type annotations with tag checks for certain types and deep traversals for others (`pyret.org`).

[2]Dart 2 implements concrete types (`dart.dev/dart-2`).

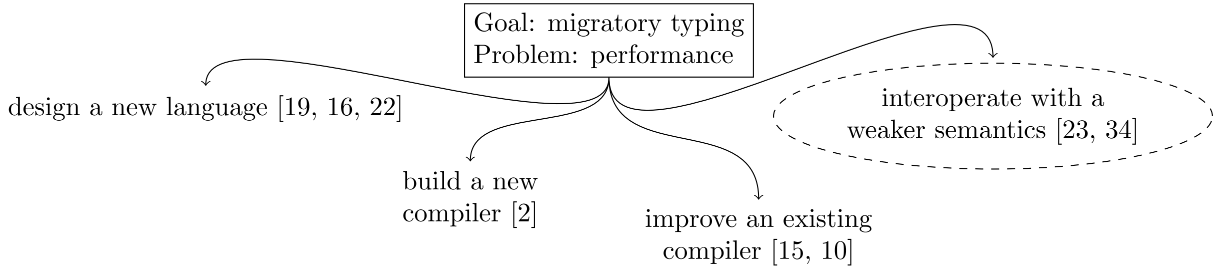

Fig. 3: Ways to influence the performance of honest migratory types.

An *erasure* migratory typing system ignores types at run-time [3, 6, 26]. Typed code benefits from static type checking, but behaves exactly the same as untyped code. Erased types therefore add zero performance overhead, do not enable type-driven optimizations, and provide zero feedback when statically-typed code receives an input that contradicts the static types. If a typed function receives a bad argument, the application proceeds without hesitation and may compute a result that is in conflict with a static type and/or the logic of the program.

Lastly, a *transient* migratory typing system partially enforces types via tag checks [33, 32]. In typed code, every elimination form and every boundary to untyped code is protected with a tag check. Each tag check matches the top-level shape of a value againt the outermost constructor of the expected type. For example, the tag check for a list of numbers accepts any list—no matter the contents. In untyped code, there are no checks. Transient types protect typed code from simple tag errors such as adding an integer to a function, but they fail to protect untyped code from lying types that are not completely checked.

The existence of different approaches indicates a conflict between the theory and practice of migratory typing. Honest types are ideal from a theoretical perspective, but require either sophisticated run-time checks or limits on the expressiveness of untyped code; figure 3 maps potential solutions to thise performance challenge. Erased types are the polar opposite, as they require no run-time support and sacrifice all guarantees. If researchers can do no better than erasure, then type-sound migratory typing is a dead end. I am not yet willing to give up, and I therefore propose to explore a compromise semantics—indicated in the right-most part of figure 3—based on a careful theoretical exploration of the design space.

2 Thesis Question

Transient types represent an exchange of guarantees for performance; however, they suffer major limitations. For one, the transient guarantees are so weak that types can mislead programmers trying to understand a faulty program. Second, the transient type enforcement strategy adds overhead to all typed code; by contrast, natural types are only expensive when typed and untyped code interact. To illustrate the differences on fully-typed programs, figure 4 compares the performance improvement in Typed Racket (natural) over untyped to the improvement in a transient Racket prototype.

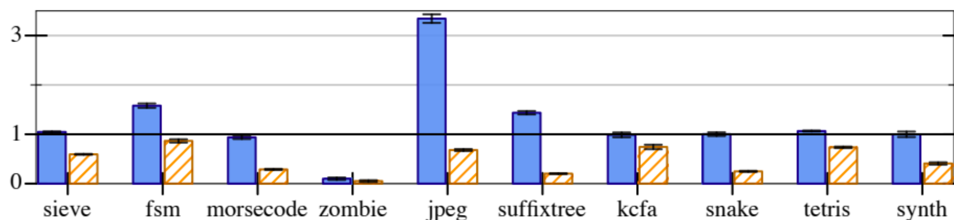

Fig. 4: Speedup factor of Typed Racket vs. untyped (solid bars) and a transient Racket vs. untyped (striped bars) [12]. Taller bars are better; bars below the 1x line indicate slowdowns.

The above brings us to the central question of my dissertation research:

Does migratory typing benefit from a combination of honest and lying types?

I plan to explore a combination of the natural and transient approaches to run-time type enforcement within one migratory typing system. In the combined system, each component in a program shall be either: dynamically-typed, statically-typed with natural type enforcement, or statically-typed with transient type enforcement. Both variants of static typing shall employ the same language of types and the same type checker. Honest types must continue to be honest and lying types must continue to be type-sound in the weakened sense described in section 3.

Beyond the dominant question, two derivative questions must be addressed. The first is *how to combine* honest and lying types in a way that preserves their respective guarantees. The second is *how to measure the performance benefit* of the combination. Section 4 outlines criteria and proposes solutions.

3 Towards a Compromise

My research to date has focused on (1) understanding the performance challenges for natural [27, 15] and (2) analyzing the design space with novel theoretical foundations [12]. These efforts have led up to the above thesis question. The performance studies motivate a combination of honest and lying types and suggest methods to validate the result. The design-space analysis provides a theoretical model for defining the combination and understanding its formal guarantees.

3.1 Performance Evaluation

Migratory typing promises the ability to mix typed and untyped code. A performance evaluation for a migratory typing system must therefore evaluate mixed-typed programs. My work proposed the first systematic evaluation method [27], showed how to compare different implementations of migratory typing [15] and adapted the method to programs with millions of ways to mix typed and untyped code [14, 15].

3.1.1 Summarizing Performance

Our method of summarizing the performance in an exponentially-large set is based on a fundamental law of software development: all programs that are not “fast enough” are equally worthless.

Takikawa et al. [27] use this relevance law to summarize the performance of a migratory typing system. If a configuration meets a fixed performance requirement, then it is good. Otherwise it is worthless. With this binary classification method, the performance of a mixed-typed program can be summarized by the proportion of configurations that meet the requirement. Likewise, a sequence of benchmarks may be summarized with a sequence of proportions. These ratios can help software developers assess the performance risk of migratory typing.

Technically, a configuration is D-deliverable if its running time is no more than Dx slower than the baseline performance with no migratory typing.[3] Given a positive number D, the proportion of D-deliverable configurations is exactly the proportion of good configurations described above.

To accomodate varying notions of good performance, Takikawa et al. [27] combine the proportions for D between 1 and 20 into a plot. The x-axis of such a plot ranges over values of D. The y-axis ranges over the number (or percent) of configurations. Figure 5, on the left-hand side, presents an example for a program with eight modules. The key takeaway is that the plot answers an important question and does not require an exponential amount of space.

Variations on the D-deliverable metric can answer similar questions about a mixed-typed program. For example, the two other plots in figure 5 relax the metric to allow one (middle plot) and two (right plot) extra type conversion steps. In this case, one conclusion supported by the right-most plot is that if a 10x slowdown is acceptable and the client is willing to add types to at most two extra modules, then 80% of the configurations are within range of an acceptable state. Once again the key takeaway is not the particular

[3] For example, in Racket, the fully-untyped configuration is an appropriate baseline. In transient Reticulated Python, the fully-untyped configuration run via Python (not via Reticulated) is an appropriate baseline because it is the starting point for a developer who wishes to migrate code [14].

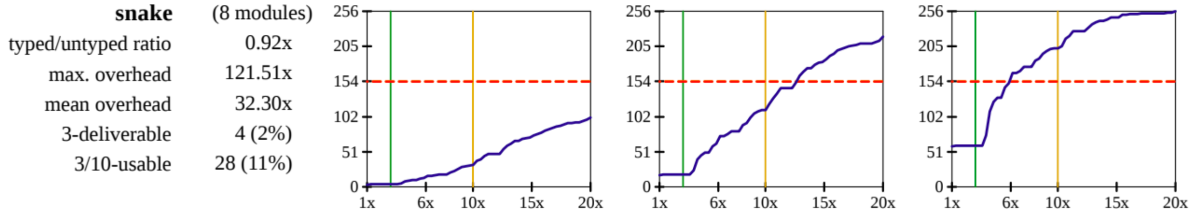

Fig. 5: Counting D-deliverable configurations in the `snake` benchmark [27]. The x-axis ranges over D values; the vertical lines mark $x = 3$ and $x = 10$. The y-axis counts configurations; the dashed horizontal line marks 60% of all configs. The thick blue line is the number of x-deliverable configurations.

conclusion, but the fact that the method helps answer practical questions about the implementation of migratory typing.

3.1.2 Comparing Implementations

The D-deliverable metric enables comparisons of different migratory typing systems. If there are two languages that can execute the same program, then the language with better performance is the one that maximizes the proportion of D-deliverable configurations.

Greenman et al. [15] use this observation to compare three versions of Racket: v6.2, v6.3, and v6.4. Racket v6.3 contains a few improvements inspired by the performance evaluation of Racket v6.2 [27]. Racket v6.4 contains many more changes: it inlines the contract checks for simple typed functions, validates struct predicates with a first-order check, and reduces the memory overhead of contracts in general. Figure 6 shows the effect of these changes on one benchmark. The curve for version 6.4 lies above the others, meaning the percent of D-deliverable configurations is larger for every value of D along the x axis.

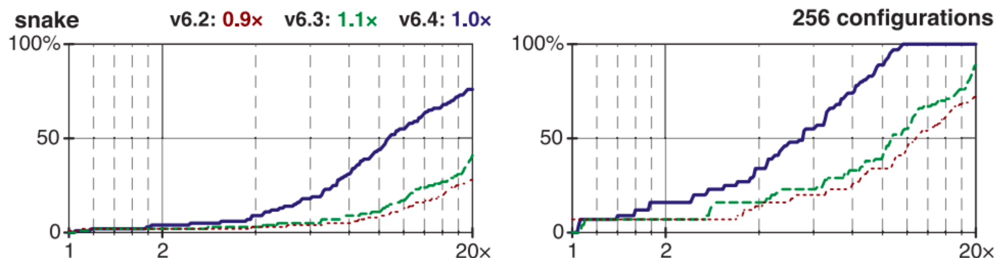

Fig. 6: Comparing performance across three versions of Typed Racket; the right plot allows one type conversion step

Similar plots have helped other researchers validate their designs:[4] Bauman et al. [2] demonstrate the benefits of adding a tracing JIT compiler to Typed Racket; Feltey et al. [10] measure the impact of collapsible higher-order contracts; and Greenman and Felleisen [12] compare Typed Racket to a prototype implementation of transient type checks.

3.1.3 Scaling the Method

Greenman and Migeed [14] adapt the method to evaluate the performance of Reticulated Python. In contrast to Typed Racket, Reticulated allows optional type annotations at a fine granularity. Every function parameter, function return type, and class field may be optionally annotated. Unfortunately for the exhaustive method, this freedom means that relatively small programs may have a huge number of configurations; counting the proportion of D-deliverable becomes impractical for a class with 20 fields.

[4] Kuhlenschmidt et al. [16] plot the D-deliverable configurations in Typed Racket alongside a count based on the overhead of a new language, Grift, relative to Racket. The latter is not the D-deliverable metric because removing all gradual typing from a Grift program does not produce a Racket program.

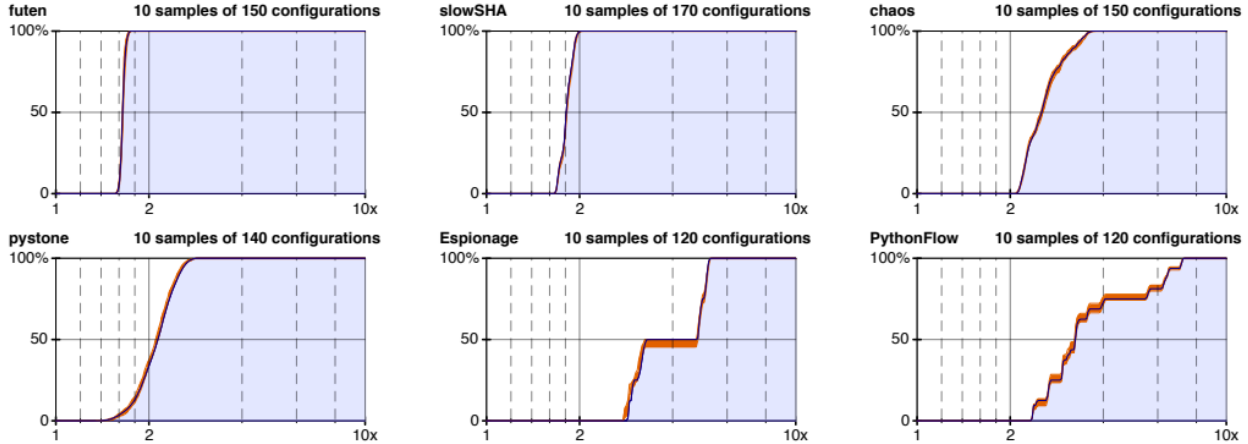

Fig. 7: Thin blue lines plot D-deliverable configurations; thick brown lines plot 95% confidence intervals

The paper demonstrates, however, that a linear number of random samples can approximate the true number of good configurations. Intuitively, the result says that the overhead experienced by N developers provides useful information for the next one to add types to the same program.

To approximate the proportion of D-deliverable configurations, first select s configurations uniformly at random and count the proportion of D-deliverable configurations in this sample. Repeat for r samples to build a set of proportions. Use the set to build a confidence interval, and finally interpret the confidence interval to approximate the true proportion of deliverable configurations.

Greenman and Migeed [14] perform an experiment in which $r = 10$ is a constant and $s = 10 * (F + C)$ is linear in the number of functions F and classes C in a benchmark. For six benchmarks with 12 to 17 functions and classes each, they generate six 95% confidence intervals. To validate these results, they collect the running time of every configuration in which: any function/method may be typed or untyped, and the set of fields for any class may be typed or untyped.[5] As figure 7 shows, the confidence intervals provide a tight bound on the ground truth.

Greenman et al. [15] validate this sampling method on Typed Racket programs. They use the same number of samples and same linear sample size and find that the intervals yield tight approximations.

3.1.4 Benchmark Suite

Greenman et al. [15] formally introduce a suite of mixed-typed benchmark programs. These GTP benchmarks are available online[6] and have been used to validate other changes to Typed Racket [12, 2].

3.2 Design Space Analysis

Models of migratory typing systems come in many varieties—often because the models reflect a proof-of-concept implementation [3, 21, 20, 1, 7, 17, 5, 34, 19, 33, 29, 8]. These models share the common goal of mixing static and dynamic typing, but realize the goal with different formalizations. Unfortunately, the diversity makes it difficult to compare properties of models in a scientific manner.

My work on comparing approaches to migratory typing employs a model that expresses realizations as different semantics for a common surface language. The common model enables well-founded comparisons of properties such as type soundness and complete monitoring. Additionally, the model facilitates the design of new semantics for migratory typing.

[5] Reticulated supports the definition of many more configurations in each program; nevertheless, the experiment suffices to validate the sampling method.

[6] `docs.racket-lang.org/gtp-benchmarks/`

3.2.1 A Spectrum of Type Soundness

Greenman and Felleisen [12] introduce a model to compare natural, erasure, and transient migratory typing as different semantics for a common surface language. The common language is mixed-typed in the style of Matthews and Findler [18]; it syntactically combines a statically-typed language with a dynamically-typed language via boundary terms. For example, the typed expression $((\mathsf{dyn}\,(\mathsf{Int} \Rightarrow \mathsf{Int})\ \lambda x_0.\, x_0)\ 2)$ applies a dynamically-typed value to a statically-typed input. The type annotation $(\mathsf{Int} \Rightarrow \mathsf{Int})$ helps the static type checker validate the application and may affect the behavior of a semantics.

In this model, the differences between type-enforcement strategies come about as different behaviors for boundary terms. The natural semantics strictly enforces types at a dynamic-to-static boundary. Incoming higher-order values get wrapped in a proxy to monitor their behavior; other values receive an exhaustive check (figure 8, left). At a static-to-dynamic boundary, the natural approach wraps outgoing higher-order values to protect against future untyped inputs. Since higher-order values may appear within first-order data structures, the latter require a traversal.

The erasure semantics treats boundary terms as a no-op. Any value may cross any boundary.

The transient semantics enforces boundary terms with tag checks (figure 8, right); however, it also treats elimination forms in typed code as boundaries. A dynamic-to-static boundary checks that the top-level shape of an incoming value matches the outermost constructor of the expected type. A static-to-dynamic boundary lets any value cross; if typed code satisfies a tag-level soundness guarantee, then such values are certain to match the outermost constructor of the expected type. Transient achieves this guarantee by guarding every elimination form in typed code with a dynamic-to-static check. Thus if the untyped value $\langle -2, 0\rangle$ enters typed code via a $(\mathsf{Nat} \times \mathsf{Nat})$ boundary and typed code projects the first element of the pair expecting a nonnegative integer, a runtime check halts the program.

$$\boxed{\mathcal{D}_{\mathsf{N}} : \tau \times v \rightarrow e}$$
$$\begin{array}{ll}
\mathcal{D}_{\mathsf{N}}(\tau_0 \Rightarrow \tau_1, v_0) & = \mathsf{mon}\,\tau_0 \Rightarrow \tau_1\ v_0 \\
\quad \text{if } v_0 \text{ is a function} & \\
\mathcal{D}_{\mathsf{N}}(\tau_0 \times \tau_1, \langle v_0, v_1\rangle) & = \langle \mathsf{dyn}\,\tau_0\ v_0, \mathsf{dyn}\,\tau_1\ v_1\rangle \\
\mathcal{D}_{\mathsf{N}}(\mathsf{Int}, i_0) & = i_0 \\
\mathcal{D}_{\mathsf{N}}(\mathsf{Nat}, i_0) & = i_0 \\
\quad \text{if } 0 \leq i_0 & \\
\mathcal{D}_{\mathsf{N}}(\tau_0, v_0) & = \mathsf{Error} \\
\quad \text{otherwise} &
\end{array}$$

$$\boxed{\mathcal{S}_{\mathsf{N}} : \tau \times v \rightarrow e}$$
$$\begin{array}{ll}
\mathcal{S}_{\mathsf{N}}(\tau_0 \Rightarrow \tau_1, v_0) & = \mathsf{mon}\,(\tau_0 \Rightarrow \tau_1)\ v_0 \\
\quad \text{if } v_0 \text{ is a function} & \\
\mathcal{S}_{\mathsf{N}}(\tau_0 \times \tau_1, \langle v_0, v_1\rangle) & = \langle \mathsf{stat}\,\tau_0\ v_0, \mathsf{stat}\,\tau_1\ v_1\rangle \\
\mathcal{S}_{\mathsf{N}}(\tau_0, v_0) & = v_0 \\
\quad \text{otherwise} &
\end{array}$$

$$\boxed{\mathcal{D}_{\mathsf{T}} : \tau \times v \rightarrow e}$$
$$\begin{array}{ll}
\mathcal{D}_{\mathsf{T}}(\tau_0 \Rightarrow \tau_1, v_0) & = v_0 \\
\quad \text{if } v_0 \text{ is a function} & \\
\mathcal{D}_{\mathsf{T}}(\tau_0 \times \tau_1, v_0) & = v_0 \\
\quad \text{if } v_0 \text{ is a pair} & \\
\mathcal{D}_{\mathsf{T}}(\mathsf{Int}, i_0) & = i_0 \\
\mathcal{D}_{\mathsf{T}}(\mathsf{Nat}, i_0) & = i_0 \\
\quad \text{if } 0 \leq i_0 & \\
\mathcal{D}_{\mathsf{T}}(\tau_0, v_0) & = \mathsf{Error} \\
\quad \text{otherwise} &
\end{array}$$

$$\boxed{\mathcal{S}_{\mathsf{T}} : \tau \times v \rightarrow e}$$
$$\mathcal{S}_{\mathsf{T}}(\tau_0, v_0) = v_0$$

Fig. 8: Boundary checks for natural (left) and transient (right)

These three methods of enforcing type boundaries lead to three different semantics for surface programs. One may compare the results of running one program via the three semantics, and one may formulate theorems that characterize general differences. The model therefore serves as a tool for design analysis. Greenman and Felleisen [12] demonstrate this point by proving three pairs of type soundness theorems for the semantics. For typed contexts, these theorems roughly guarantee the following: the natural semantics can only yield values that fully match the expected type; the erasure semantics can yield any value; and the transient semantics can only yield values with a tag that matches the outermost constructor of the expected type. Sibling theorems describe the behavior of untyped code.

Additionally, the model serves as a tool for language design. One may develop a new semantics by proposing a new strategy for checking the boundaries between typed and untyped components. Greenman and Felleisen [12] present two such variants of the natural semantics, dubbed co-natural and forgetful, that bridge the gap between natural and transient. Co-natural allocates wrappers for all kinds of structured data—

211 not only higher-order values—and thereby reduces the amount of checking at a boundary to a tag check.
212 Forgetful extends co-natural; if a wrapped value reaches a boundary, a new wrapper replaces the existing
213 one.[7] Both co-natural and forgetful provide a type soundness guarantee that resembles type soundness for
214 natural. They both fail, however, to detect some type violations that natural detects. Technically, co-natural
215 detects more errors than forgetful, which detects more errors than transient [12]. Greenman et al. [13] present
216 a variant of transient (Amnesic) that provides more precise blame information when a boundary error occurs.

217 **3.2.2 A User Study**

218 Tunnell Wilson et al. [31] use the model of Greenman and Felleisen [12] to create a survey about semantics
219 for mixed-typed languages. The survey employs the common surface syntax and three semantics: natural,
220 erasure, and transient. Each question presents one program and two or three possible outcomes of running
221 the program. Repondents must form an opinion based on two *attitudes*: (1) do they like/dislike the behavior,
222 and (2) do they find it expected/unexpected. The two attitudes form a matrix of four possible answers.

223 The authors administered their survey to three populations: software engineers at a major Silicon Valley
224 technology company; computer science students at a highly selective, private US university; and Mechanical
225 Turk workers with some programming experience. They found a preference for the natural semantics—and
226 more generally, for enforcing all claims implied by the types—across all three populations.

227 **3.2.3 Honest vs. Lying Types**

228 The different type soundness theorems for natural and transient demonstrate their different guarantees for
229 typed code. Clearly, natural-typed code can trust full types and transient-typed code can trust top-level
230 type constructors. Type soundness fails, however, to describe their different guarantees for untyped code.
231 Suppose that one untyped component E expects a pair of numbers from a typed component; suppose further
232 that the typed component provides a pair that it receives from a different untyped component—and that
233 the pair contains strings rather than integers.

$$E[\mathsf{stat}\,(\mathsf{Int}\!\times\!\mathsf{Int})\ (\mathsf{dyn}\,(\mathsf{Int}\!\times\!\mathsf{Int})\ \langle\texttt{'hello'}, \texttt{'world'}\rangle)]$$

234 A natural semantics halts the program when the pair reaches the boundary to typed code. A transient
235 semantics lets the pair cross into typed code and out again to the untyped client. Both behaviors are
236 permitted by type soundness. After all, soundness for untyped code has nothing to say about the type of a
237 result value. But the transient behavior means that untyped code cannot trust a typed API.

238 In general, a natural type is *honest* because it is a valid claim about all future behaviors of a value.
239 If natural accepts a value at a certain type, then it has either fully-checked the value or wrapped it in a
240 monitoring proxy. A transient type is valid in one specific context, and *lying* to the rest of the program. For
241 example, the type ($\mathsf{Int}\!\times\!\mathsf{Int}$) above is only enforced in the visible typed component; the rest of the program
242 cannot assume that the type is a valid claim about the contents of pairs that flow through the component.

243 Greenman et al. [13] formalize these intuitions with a complete monitoring theorem. Complete monitoring
244 comes from prior work on higher-order contracts; in brief, a contract system satisfies complete monitoring
245 if every channel of communication between components can be monitored by a contract [9]. A mixed-typed
246 language satisfies complete monitoring if runtime checks protect all communications across boundaries. If
247 so, then a pair value cannot transport a string such as `'hello'` across a boundary that expects a number.

248 The key to proving a complete monitoring theorem is to enrich the syntax of the model with ownership
249 annotations. Ownership annotations state which components are responsible for the expressions and values
250 in a program. When a value flows across a boundary, its ownership changes depending on the type checks
251 that occur. If the checks fully validate the boundary type, the value replaces its previous owners with a new
252 one. Otherwise, the value keeps the previous owners and gains a new owner. In this framework, a semantics
253 satisfies complete monitoring iff it never lets a value accumulate more than one owner.

254 4 Research Challenges

255 Three challenges stand between the thesis question and an answer:

[7] The forgetful semantics and its type soundness are inspired by forgetful contracts [11].

256 1. Combine honest and lying migratory typing in a model; formulate and prove safety properties.
257 2. Implement the model for Racket; re-use the Typed Racket type system.
258 3. Evaluate the performance of the combined semantics.

259 4.1 Challenge 1: Model

260 The first challenge is to extend our semantic model of migratory typing to combine honest and lying types
261 in one semantics. The current model supports them in parallel developments; the task is to allow interoper-
262 ability. The new model must allow the definition of honest-typed code, lying-typed code, and untyped code
263 in the surface syntax. All three must be able to share all kinds of values via boundary terms. In particular,
264 sharing implies that lying code must accept monitored values from honest code. Honest-typed code may
265 need a rewriting pass in the style of transient.

266 The primary goal of the model is to state and prove safety properties for each of the three languages
267 in the model. Honest types must be trustworthy in any context; they must satisfy a complete monitoring
268 theorem and a standard type soundness theorem. Lying types must match the type-tag of values. Untyped
269 code must have well-defined behavior.

270 A secondary goal of the model is to minimize the amount of run-time checking that is needed to ensure
271 safety. Lying-typed code may be able to leverage the properties of honest types to avoid some overhead.
272 Honest-typed code may benefit from trusting the constructors of lying-typed values. The challenge is to
273 explore the design space, find methods that are likely to give a performance benefit, and (time permitting)
274 pursue a full-fledged implementation.

275 The model must scale to union types, universal types, and recursive types. That is, these types must
276 either be part of the model or else it must be clear how to add them—both for static typing and for run-
277 time checks. Greenman and Felleisen [12] describe how to support such types in transient and present an
278 implementation that does so, but integration with natural types may pose new challenges.

279 4.2 Challenge 2: Implementation

280 The second challenge is to validate the model through an implementation. Racket is a natural target for
281 such an implementation, because it supports honest migratory typing and partially supports lying migratory
282 typing. What remains is to extend the partial support for lying migratory typing and to combine the two
283 strategies according to the model. Honest-typed code must continue to use the type-driven optimizer [25]
284 and to protect itself against untyped code. Lying-typed code must be able to share type definitions with
285 honest-typed code and values with both honest-typed and untyped code. Time-permitting, I may explore
286 further extensions.

287 4.2.1 Primary Goals

288 • Extend the current partial support for lying migratory typing to accomodate all Typed Racket types
289 that appear in the functional GTP benchmarks. Each type needs a matching tag-check. Some tag-
290 checks are easy to define; for example, the proper tag-check for the `Symbol` type is the `symbol?`
291 predicate. Other types present a choice: should the check for `Listof` ensure a proper list? should the
292 check for `->*` validate arity and keyword arguments? I plan to initially answer "yes" to both questions
293 and generally to check all possible first-order properties; at least until there is evidence that these
294 checks are too expensive.

295 • Avoid the Racket contract library because contract combinators have administrative overhead. Tag-
296 checks must be realized with simple Racket code wherever possible to improve run-time performance
297 and take advantage of compiler optimizations.

298 • Interact safely with Typed Racket. Statically, lying and honest-typed code must be able to share type
299 definitions. Dynamically, honest-typed code must protect itself against lying values.

300 • Provide relatively fast compilation times for lying-typed Racket. Some overhead relative to honest-
301 Typed Racket may arise from the pass that rewrites typed code. Anything more than a 10% slowdown,
302 however, must be studied and explained.

4.2.2 Secondary Goals

- Add support for class and object types. These types need a tag check, but there are many question about how such checks should explore compatible values. One idea is to mimic the first-order checks done by the contract system; the question is whether those checks suffice for soundness.

- Investigate a static analysis to remove tag checks. Typed Racket employs occurrence typing to propagate information about type-tests. A tag-check is a simple type test, and the success of one check has implications for the rest of the program. For example, if a block of code projects an element of an immutable pair twice in a row, then only the first projection requires a tag check.

- Adapt the Typed Racket optimizer. The current implementation of lying types is incompatible with the Typed Racket optimizer. For one, the implementation outputs code that the optimizer cannot handle. More significantly, some optimization passes are inappropriate for lying-typed code because they rely on honest type information. Lying-typed code may still benefit from simple optimizations, however, so it is worthwhile to try reusing the optimizer. For example, it may specialize an application of `+` to expect unboxed numbers.

4.3 Challenge 3: Evaluation

The third challenge is to test the hypothesis that a combination of honest and lying types is better than either one individually. There are a few ways that a programmer could benefit: changing one module from honest to lying may reduce the cost of type boundaries, changing a collection of modules from lying to honest may remove many tag checks, and changing a library from honest to lying may improve the performance of untyped clients.

I plan to use the GTP benchmarks in a systematic performance evaluation. It is unclear, however, how to conduct such an evaluation. An exhaustive study requires 3^N measurements for each program with N modules, which is a large amount of data to collect and interpret. I propose two alternatives for now.

To test a library, the existing method suffices. One can convert the library to be lying-typed and measure an honest-typed performance lattice.

To test the benefits for program authors, who now have three choices for every module, I propose a path-based metric based on two assumptions. First, I assume that authors are seeking fully-typed programs. Second, I assume that the authors are seeking honest-typed programs. For the evaluation, the question is what percentage of paths through the honest-typed performance lattice have no configurations that exceed a certain overhead, say 10x, given the option at each step of converting some modules to lying types. The idea is that a programmer moves up the lattice by adding type annotations to one module at a time. If, at any step, honest types lead to unacceptable overhead, the programmer can switch to lying because both use the same static types. Once the programmer has added "enough" types, switching back to honest should offer a performance boost (in addition to the stronger guarantees of honest types).

Both assumptions above threaten the validity of any conclusions drawn from the evaluation. Experience with Racket suggests that fully-typed programs are not the norm. Programmers often end up with partially-typed programs, especially when they intend to support typed and untyped client programs. It is also unclear that honest-typing offers the best performance for fully-typed programs. There are 2^N ways of mixing honest and lying typing in a fully-typed program; without a full evaluation, one cannot be sure that a mixed program out-performs the honest version.

5 Proposed Schedule

There are four major tasks ahead: develop a model, build an implementation, evaluate the implementation, and write a dissertation. I additionally plan to write a research paper about the model and implementation.

Work on the model can begin immediately. By the end of this year, it should be clear whether the model can support an implementation. Implementation work must begin with a review of the existing lying-typed Racket system [12]; I have already started this update, and plan to continue working in parallel with the modeling. Evaluations will begin as soon as the implementation can support simple benchmarks. These preliminary measurements shall inform the protocol for a final performance evaluation. The evaluation will

```
Sep                       + implementation                                              .
.                         |                                                             .
Oct         + model       |                                                             .
.           |             |                                                             .
Nov         |             |                                                             .
.           |             |                                                             .
Dec         |             |                                                             .
.           |             |                                                             .
Jan’20      |             |                                                             .
.           +             |                                                             .
Feb                       |                                                             .
.                         |             + evaluation                                    .
Mar                       |             |                                               .
.                         |             |                                               .
Apr                       |             |                                               .
.                         +             |             + paper                           .
May                                     |             |                                 .
.                                       |             |             + dissertation      .
Jun                                     +             +             |                   .
.                                                                   |                   .
Jul                                                                 |                   .
.                                                                   |                   .
Aug                                                                 |                   .
.                                                                   +                   .
```

Fig. 9: Proposed schedule

take several weeks to finish; as the evaluation is running, I plan to write a research report on the combined system. The dissertation will combine this report with my other findings on migratory typing. I plan to defend my dissertation during the Fall 2020 semester.

References

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

[2] Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. 2017. Sound Gradual Typing: Only Mostly Dead. *PACMPL* 1, OOPSLA (2017), 54:1–54:24.

[3] Gavin Bierman, Martin Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP*. 257–281.

[4] Gavin Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding Dynamic Types to C#. In *ECOOP*. 76–100.

[5] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. 2016. Practical Optional Types for Clojure. In *ESOP*. 68–94.

[6] Gilad Bracha and David Griswold. 1993. Strongtalk: Typechecking Smalltalk in a Production Environment. In *OOPSLA*. 215–230.

[7] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levy. 2017. Fast and Precise Type Checking for JavaScript. *PACMPL* 1, OOPSLA (2017), 56:1–56:30.

367 [8] Benjamin W. Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. 2018. KafKa: Gradual Typing for Objects. In
368 *ECOOP*. 12:1–12:23.

369 [9] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In
370 *ESOP*. 214–233.

371 [10] Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. Collapsible
372 Contracts: Fixing a Pathology of Gradual Typing. *PACMPL* 2, OOPSLA (2018), 133:1–133:27.

373 [11] Michael Greenberg. 2015. Space-Efficient Manifest Contracts. In *POPL*. 181–194.

374 [12] Ben Greenman and Matthias Felleisen. 2018. A Spectrum of Type Soundness and Performance. *PACMPL* 2, ICFP (2018),
375 71:1–71:32.

376 [13] Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2019. Complete Monitors for Gradual Types. *PACMPL* 3,
377 OOPSLA (2019), 122:1–122:29.

378 [14] Ben Greenman and Zeina Migeed. 2018. On the Cost of Type-Tag Soundness. In *PEPM*. 30–39.

379 [15] Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen.
380 2019. How to evaluate the performance of gradual type systems. *JFP* 29, e4 (2019), 1–45.

381 [16] Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. Toward Efficient Gradual Typing for Struc-
382 tural Types via Coercions. In *PLDI*. 517–532.

383 [17] Andre Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. 2015. A Formalization of Typed Lua. In *DLS*.
384 13–25.

385 [18] Jacob Matthews and Robert Bruce Findler. 2009. Operational Semantics for Multi-Language Programs. *TOPLAS* 31, 3
386 (2009), 1–44.

387 [19] Fabian Muehlboeck and Ross Tate. 2017. Sound Gradual Typing is Nominally Alive and Well. *PACMPL* 1, OOPSLA
388 (2017), 56:1–56:30.

389 [20] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual
390 Typing for TypeScript. In *POPL*. 167–180.

391 [21] Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. 2013. The Ruby Type Checker. In *SAC*.
392 1565–1572.

393 [22] Gregor Richards, Ellen Arteca, and Alexi Turcotte. 2017. The VM Already Knew That: Leveraging Compile-Time
394 Knowledge to Optimize Gradual Typing. *PACMPL* 1, OOPSLA (2017), 55:1–55:27.

395 [23] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete Types for TypeScript. In *ECOOP*. 76–100.

396 [24] Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2019. Transient Typechecks are (Almost) Free. In
397 *ECOOP*. 15:1–15:29.

398 [25] Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. 2012. Typing the Numeric Tower. In
399 *PADL*. 289–303.

400 [26] Guy L. Steele, Jr. 1990. *Common Lisp the Language* (2nd ed.). Digital Press.

401 [27] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual
402 Typing Dead?. In *POPL*. 456–468.

403 [28] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: from Scripts to Programs. In *DLS*. 964–974.

404 [29] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *POPL*. 395–406.

405 [30] Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent
406 St-Amour, T. Stephen Strickland, and Asumu Takikawa. 2017. Migratory Typing: Ten years later. In *SNAPL*. 17:1–17:17.

407 [31] Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. 2018. The Behavior of Gradual
408 Types: A User Study. In *DLS*. 1–12.

409 [32] Michael M. Vitousek. 2019. *Gradual Typing for Python, Unguarded*. Ph.D. Dissertation. Indiana University.

410 [33] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big Types in Little Runtime: Open-World Soundness
411 and Collaborative Blame for Gradual Type Systems. In *POPL*. 762–774.

412 [34] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Ostlund, and Jan Vitek. 2010. Integrating Typed
413 and Untyped Code in a Scripting Language. In *POPL*. 377–388.