The Dynamic Practice and Static Theory of Gradual Typing (Accepted Draft)

Michael Greenberg
Pomona College, Claremont, CA, USA
http://www.cs.pomona.edu/~michael/
michael@cs.pomona.edu

Abstract

We can tease apart the research on gradual types into two 'lineages': a pragmatic, dynamic-first lineage and a formal, type-theoretic, static-first lineage. The dynamic-first lineage's focus is on taming particular idioms—'pre-existing conditions' in untyped programming languages. The static-first lineage's focus is on interoperation and individual type system features, rather than the collection of features found in any particular language. Both appear in programming languages research under the name "gradual typing", and they are in active conversation with each other.

What are these two lineages? What challenges and opportunities await the static-first lineage?

2012 ACM Subject Classification Social and professional topics → History of programming languages; Software and its engineering → Language features

Keywords and phrases gradual typing, implementation, challenge

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Acknowledgements I thank Sam Tobin-Hochstadt and David Van Horn for their hearty if dubious encouragment. Conversations with Ron Garcia, Matthias Felleisen, Robby Findler, and Spencer Florence improved the argumentation.

1 A tale of two gradualities

> It was the best of types, it was the worst of types,
> it was the age of static guarantees, it was the age of blame,
> it was the epoch of implementations, it was the epoch of core calculi,
> it was the season of pragmatism, it was the season of principles.
>
> —with apologies to Charles Dickens

In 2006, the idea of gradual typing emerged in two papers. Tobin-Hochstadt and Felleisen introduced the idea of mixing untyped and typed code in way such that "code in typed modules can't go wrong" using *contracts* [58, 23]; Siek and Taha showed how to relax the simply typed lambda calculus (plus some extensions) to allow for unspecified "dynamic" types to be resolved at runtime via *casts* [49].[1]

In these two papers, two parallel lines of research on gradual typing began with quite different approaches. Sam Tobin-Hochstadt summarized the distinction as 'type systems for existing untyped languages' [Tobin-Hochstadt and Felleisen] and 'sound interop btw typed and untyped code' [Siek and Taha].[2] I draw slightly different lines, identifying one lineage as being "dynamic-first" and the other as "static-first". That is: one can think about taking a d dynamic language and building a type system for it, or one can think about taking a statically typed language and relaxing it to allow for dynamism.

[1] Flanagan showed how to use a similar cast framework to relax a fancy subset type system [24]. There must have been something in the water.

[2] https://twitter.com/samth/status/1039707471290478595

The division between these two approaches is still evident, and the latter approach has an opportunity for interesting new discoveries from proof-of-concept (and more serious) implementations.[3]

Disclaimer: I have made an effort to be thorough but not comprehensive in my citations. Readers looking for a comprehensive survey will enjoy Sam Tobin-Hochstadt's "Gradual Typing Bibliography".[4] Even so, I make general claims about trends in gradual types. I try to mention the inevitable exceptions to my generalizations, but I may have missed some.

1.1 The dynamic-first approach

Tobin-Hochstadt and Felleisen use a 'macro' approach, where the unit of interoperation is the module. They are directly inspired by Racket's module system. They see the dynamic language as being somehow primary, with a static layer above:

> First, a program is a sequence of modules in a safe, but dynamically typed programming language. The second assumption is that we have an explicitly, statically typed programming language that is [a] variant of the dynamically typed language. Specifically, the two languages share run-time values and differ only in that one has a type system and the other doesn't. [58]

Their paper takes an "expanded core calculus" approach, defining an extension of the lambda calculus with a notion of module (untyped, contracted, or typed).

Dynamic-first gradual typing is about accommodating particular programming idioms in programs that allow legacy untyped code to interoperate with the newly typed fragment. Typed Racket is the first and perhaps canonical example, though TypeScript's various dialects, Dart, DRuby/Rubydust/rtc, Clojure's `specs`, Gradualtalk, Reticulated Python, and Thorn are all comparable efforts in the research community [59, 25, 7, 45, 44, 5, 66, 9]. These languages all share an approach going back chiefly to the 1990s but also earlier: we have a dynamic language and we'd like the putative benefits of static typing (for, e.g., maintenance, documentation, performance) [15, 57, 13, 16, 34].

> Not having to support a static type system frees dynamic languages up for including powerful dynamic features which would be problematic in a typed setting, e.g., reflective behaviour and metaprogramming facilities. Commonly available reflective mechanisms include support for checking available fields/methods, adding and removing fields/methods, without the need to restart or rebuild the running program, and runtime code generation. A common use of reflection is to extract all methods prefixed "test" in unit test frameworks, but also to generate names of attributes from program input. [4]

The type systems used in the dynamic-first approach tend to the unfamiliar, with features designed to capture particular language idioms, like occurrence typing, 'like' types, severe relaxations of runtime checking disciplines to avoid disrupting reference equality, and ad hoc rules for inferring particular types (e.g., telling the difference between a tuple and an array).

---

[3] There are two other distinctions one could make. First, the macro/micro distinction from Takikawa et al. and Greenman et al. [55, 33]; second, the latent/manifest distinction [29, 30]; and third, the distinction between programs whose static semantics influences their runtime (e.g., type classes) and those languages where types can be erased. These distinctions are important but less salient for my argument.

[4] `https://github.com/samth/gradual-typing-bib`

1.2 The static-first approach

Siek and Taha take a 'micro' approach, where the unit of interoperation is the expression [49]. They are inspired by Thatte's quasi-static typing and Oliart's algorithmic treatment thereof [57, 42]. While they imagine migrating programs from dynamic to static—would one ever want to go the other way?—they implicitly see the type system as primary, and gradual types as a relaxation. In their contributions:

> We present a formal type system that supports gradual typing for functional languages, providing the flexibility of dynamically typed languages when type annotations are omitted by the programmer and providing the benefits of static checking when function parameters are annotated.

Their paper does not, however, identify any particular dynamic idioms that their static type discipline disallows. Such an example might serve as motivation for wanting to relax the type system, either to accommodate existing dynamic code that uses hard-to-type idioms (e.g., as in Takikawa et al. [56]) or to write new code that goes beyond their system (e.g., as in Tobin-Hochstadt and Findler [60]). (They don't observe as much, but adding the dynamic type does add a new behavior—nontermination [1].) The code of their two lambda calculus interpreters is identical (their Figure 1); only the type annotations change. According to Siek et al.'s refined definition [52], gradual typing "provides seamless interoperability, and enables the convenient evolution of code between the two disciplines"; it is critical to their conception of gradual typing that it "relates the behavior of programs that differ only with respect to their type annotations". Lacking particular dynamic idioms to accommodate, the examples in static-first papers tend to be toy snippets mixing static and dynamic code to highlight this interoperation, even when pointing out the oversight (e.g., Section 6 from Garcia and Cimini [26]).

Work in the 'static-first' lineage cite interoperation as a motivation, not only in Siek and Taha's seminal paper [49] but especially in Wadler and Findler [68]. Later papers take interesting type feature X and show how to relax the typing rules, resolving static imprecision with dynamic checks: objects [50], polymorphism [2, 3, 63], typestate [69], information flow control [20, 22, 61], ownership types [47], effects [8], session types [35], etc. The process of relaxation was made beautifully concrete in Garcia, Clark, and Tanter's "Abstracting Gradual Typing" (AGT) [27]; Matteo Cimini and Jeremey Siek built the Gradualizer, a tool for automatically turning a type system gradual [19].

The type systems in the static-first lineage tend to look much more like those found in the conventional types literature... unsurprising, in light of AGT! The resulting theories are typically conservative extensions of their original system—statically typed programs remain acceptable—satisfying the static gradual guarantee (reducing type precision retains typeability) [52]. Many systems also enjoy the dynamic gradual guarantee (reducing type precision retains successful runs of the program), though notably not for several type systems implementing hyperproperties [62, 64].

2 Dynamic trouble in static paradise

> It is easy to design a type system, and it is reasonably straightforward to validate some theoretical property. However, the true proof of a type system is a pragmatic evaluation. To this end, it is imperative to integrate the novel ideas with an existing programming language. Otherwise it is difficult to demonstrate that the type system

> accommodates the kind of programming style that people find natural and that it serves its intended purpose.
>
> To evaluate occurrence typing rigorously, we have implemented Typed Scheme.
>
> —Tobin-Hochstadt and Felleisen [59]

2.1 A distinction without a difference?

Does it matter whether one starts from dynamic typing and works up to static or starts with static relaxes to allow dynamic typing?[5] Only the dynamic-first lineage addresses particular examples and the particular difficulties they introduce into the resulting systems.

Dynamic-first gradual typing is motivated by particular, existing legacy code in particular, existing languages. Whatever theory dynamic-first systems come up with must be accommodated to the host language's pre-existing conditions.

> [D]ynamic language programmers often employ programming idioms that impede precise yet sound static analysis. For example, programmers often give variables flow-sensitive types that differ along different paths, or add or remove methods from classes at run-time using dynamic features such as reflection and `eval`. [7, 46]

Static-first gradual typing typically lacks such motivation, studying interoperation more abstractly. Static-first gradual typing often studies type system *features* without any attempt to accommodate the idiosyncrasies of any particular, existing programming language. (There are, of course, laudable exceptions [47, 6]

The distinction becomes clear when we see what is actually implemented: the overwhelming majority of the existing implementations of gradual typing start with a dynamic language and grow an appropriate type system for it. There are two notable exceptions: Nom and Grift are direct implenmentations of the static-first theory [40, 36]; C# is a statically typed language which grew a dynamic runtime unrelated to the theory of gradual types.

It is surprising that the theory should take a static-first approach, but the practice takes a dynamic-first one. It would seem that nobody has tried to apply the static-first theory to a pre-existing statically typed language. A set of concrete, desirable dynamic idioms would allow the dynamic-first and static-first lineages to address the same challenges and benefit more from each other's insights. I offer one such challenge in detail, followed by some higher level challenges (Section 3).

2.2 A dynamic idiom: `flatten`

A canonical example of a dynamic programming idiom is the `flatten` function (Figure 1). The `flatten` function takes arbitrarily nested lists (formed by `cons` cells) and produces a single flat list containing all of the elements in a left-to-right traversal. Thinking of such nested lists as trees, `flatten` computes the fringe of the tree. The `flatten` function works because there are predicates `null?` and `cons?` of conceptual type $? \to$ `bool`. While it is a perfectly safe function—nothing in it can go wrong at runtime—it is hard to assign a type to `flatten`, since the type of heterogenous nested lists cannot be written down in simple type languages. Like in Tobin-Hochstadt and Findler's "gradual typing poem" [60], we assign the dynamic type to patch over a programming idiom that our type system cannot account for (there, cyclic data structures; here, heterogeneity and arbitrary nesting).

[5] `https://twitter.com/lambda_calculus/status/1039702266679369730`

```
(define (flatten x)
  (cond
    [(null? x) '()]
    [(cons? x) (append (flatten (car x)) (flatten (cdr x)))]
    [else      (list x)]))
```

```
> (flatten '(1 (2 3) (((4) (5)) (6 7 8 (9)))))              ; example
'(1 2 3 4 5 6 7 8 9)
```

Figure 1 The `flatten` function in Scheme/Racket

2.2.1 `flatten` in dynamic-first gradual typing

Occurrence typing captures the reasoning in `flatten` perfectly, allowing Typed Racket to infer the type of flatten without any annotations.[6]

Occurrence typing is not a standard type system feature. It is not even a particularly desirable one according to the tastes of the static typing community, as evidenced its lack of adoption in the there. Folks who like static types seem to prefer dependent pattern matching for flow-sensitive reasoning. Occurrence typing is used in Typed Racket because it works: it "accommodates ... modes of reasoning ... programmers use"—Typed Racket was designed "to support Scheme idioms and programming styles" [59].

2.2.2 `flatten` in static-first gradual typing

How might one write `flatten` in the static-first lineage? While `flatten` is an example of a simple dynamic idiom, the static-first lineage has only recently devised systems that can accommodate it. Most static-first gradual type systems don't offer type tests, though there are noteworthy exceptions [37, 38, 12]. Siek and Tobin-Hochstadt's true union types [51] can handle the definition at the same moral type of $? \to$ `list` $?$ (in their notation, $\star \to \mu X.\texttt{unit} \cup \star \times X$). Toro and Tanter can't quite handle it [65]. Recent work by Castagna, Lanvin, and others might be able to accommodate the idiom, as well [17, 18].

3 An opportunity

Static-first gradual typing has the opportunity to (a) identify the particular new programs gradual typing allows us to write or interoperate with and (b) verify that we can implement gradual type systems accommodating these new programs.

Enumerating concrete examples and implementing the theory will stress-test our understanding of the theory, leading to refinements and improvements in theory and practice.

3.1 Gradual typing for expressiveness

> For any interesting programming language, there will always be some programs that [the] user must rewrite to accommodate a static type checker.

[6] Typed Racket assignes the type `(-> Any (Listof Any))`. It is reasonable that Typed Racket cannot express the negated union lurking in the codomain under the `Listof`, where one might want to write `(-> Any (Listof (- Any (Listof Any))))`. Recent work combining gradual typing with set theoretic types might be able to express this more precise type [18].

—Mike Fagan [21]

If one studies gradual typing in order to be able to write new kinds of programs, I offer three examples of dynamic idioms that might serve as motivating examples.

1. **Heterogeneous structures.** While `flatten` is a "toy" function, it makes non-trivial use of type predicates in a way that is simultaneously realistic and challenging to existing static-first type theories.[7] Put another way, I might want to temporarily "cheat" and view my structured data a little less formally than the type system would ordinarily allow. How can such shenanigans be safely accommodated in languages that want types to mean things? What does that mean for more complicated structures like sets and maps that, e.g., compare values to maintain invariants?
2. **Semi-structured data, like JSON, YAML, and XML.** Even when these formats don't take advantage of recursion, they represent heterogeneous data that isn't easily accommodated by type systems.
3. **Attaching information to HTTP request and response objects.** So-called "middleware" in web servers is typically implemented as a quasi-continuation-passing function $\texttt{mw} : \texttt{Req} \times \texttt{Resp} \times (\texttt{unit} \to \alpha) \to \alpha$, where `Req` and `Resp` are (mutable) HTTP request and response objects and the third argument is a (thunked) continuation.

   The middleware function `mw` can look up user information and then *attach* that user information to the request object, making it available for later processing.

3.2 Gradual typing for interoperation

If one studies gradual typing in order to be able to interoperate programs from different idioms, what better way to show it than by implementing an interoperation library for, say, OCaml and Python or Haskell and Julia or Scala and Clojure?

There are several challenges left unaddressed by theoretical treatments of interoperation.

1. **Numerics.** Dynamic languages typically have a "numeric tower" with rules for when values move from more precise types (e.g., unbounded bignum integers or precise rationals) to less precise ones (e.g., fixed or floating point numbers). Statically typed languages typically require explicit coercions (e.g., `fromIntegral` in Haskell) and sometimes have separate operations for each numeric type (e.g., `+` and `+.` in OCaml).

   For static languages to interoperate with dynamic ones, the promotion rules will leak. A statically polymorphic function run in the dynamic side could result in a promotion, which might violate parametricity. These thorny questions have been studied for Racket's complicated numeric tower already [53]; what should happen in other settings?
2. **Data structures, interfaces vs. translations, and guarantees.** Tobin-Hochstadt and Felleisen assume that "the two languages share run-time values and differ only in that one has a type system and the other doesn't" [58]. This will not generally hold. The representation of Racket and OCaml strings are different, but so are their interfaces: string constants in Racket are immutable,[8] while OCaml's are mutable.

   When we move a value from language A to language B, we may want to send it over as an object with an interface—allowing B to use the object with A's semantics—or to map it to one of many possible targets in B. Such translations will come with a computational

[7] Fagan's PhD thesis is rich in such examples [21].

[8] Though not all strings are immutable; see `https://docs.racket-lang.org/reference/strings.html`.

cost—typically linear but sometimes worse!—but allow several benefits: it may be more efficient to avoid the A/B language barrier, B may have more efficient representations in general, and B may provide guarantees that A does not.

3. **Type-driven features.** Muehlboeck and Tate have shown that a variety of type-based features in C# lead to violations of the dynamic gradual guarantee [40]. Haskell's type classes are a challenging related feature, not just for numerics, but for determining which monad is run by, e.g., a `do` block. How might Haskell mix with dynamic code that performs IO or other effects? How might dynamic values in Haskell enjoy the `Ord` instances necessary to build, e.g., heterogeneous sets?
4. **Minimizing annotation overhead.** Static-first gradual typing typically studies elaborated core calculi—many papers do not describe the surface language that generates the casts. How can we minimize the annotation overhead? What cast insertion strategies are appropriate? (Swamy et al. give a starting point [54].) What tool support do we need—inference [48], something more exploratory, along the lines of Campora et al. [14], or more tools for eliminating checks [41]?

   The idea of minimizing annotation overhead is implicit in gradual versions of fancy type systems, where the "dynamic" side is a typical static type system and the "static" side is a fancier type system (e.g., information flow [20, 22, 61]). Experiments with an implementation are a natural next step.
5. **Garbage collection.** Who is responsible for allocating and deallocating? When does each language's GC run?
6. **Linking.** What is the right object/header format? It is a shame that if we were to try to link Rust and Haskell, we would probably have to go through a C API!
7. **Debugging.** How does one take the hodgepodge of stack frames, thunks, and continuations from mixing two real languages and produce something intelligible?

3.3 In which I am gravely mistaken

"No, no," you say, "that's not right. We can already do all of this!" I've enumerated a selection of objections below.

1. **Static languages can accommodate those idioms.** You can just *make* a datatype for JSON; OCaml already has s-expression support instead of the general dynamic type; Haskell has dependency, `Data.Dynamic` and `Type.Reflection`, and the Aeson library.

   Response: Maybe the static world never really wanted to interoperate with dynamic types. But there are still challenges: Yesod's middleware only supports textual data on sessions, so dynamic frameworks still have an edge.

   Type-based programming in Haskell is strong medicine, and every project has a limited complexity budget. Not everyone wants to spend their complexity budget on types. For example, the `flatten` function can be written in Haskell (Figure 2), but it is somewhat less readable than its Racket counterpart.
2. **We can use linking types.** Patterson and Ahmed's linking types solve this problem [43].

   Response: Let's implement it! Linking types have been successful for proving things about translations [11, 10]. Do they have any bearing on implementations? Work by Matthews et al. offers some gestures in this direction [38, 37]; Gray et al. [28] offer a substantial interface between two very different languages (Java and Scheme).
3. **These ideas are already implemented.** GradualTalk, DRuby/rtc/Rubydust, Reticulated Python, Nom and Grift are implementations [5, 25, 7, 45, 66, 40, 36]; some theoretical work offers web interfaces for experimentation with their type theory [62].

```
{-# LANGUAGE GADTs, TypeApplications #-}
import Data.Dynamic
import Type.Reflection

kindStar = typeRepKind (typeRep @Bool)
listCon = typeRep @[]
dynamicTypeRep = typeRep @Dynamic

flatten :: [Dynamic] -> [Dynamic]
flatten [] = []
flatten (dx@(Dynamic typeRep x):dxs) =
  let x' = case eqTypeRep (typeRepKind typeRep) kindStar of
             Just HRefl ->
               case typeRep of
                 App ctor arg ->
                   case eqTypeRep ctor listCon of
                     Just HRefl ->
                       case eqTypeRep arg dynamicTypeRep of
                         Just HRefl -> flatten x
                         Nothing ->
                           withTypeable arg (flatten (map toDyn x))
                     Nothing -> [dx]
                 _ -> [dx]
             Nothing -> [dx] in
  x' ++ flatten dxs
```

Figure 2 A version of `flatten` using Haskell's `Dynamic` type

Response: Let's do more! Let's scale them to real, existing languages; let's implement the various challenge problems I've described.

GradualTalk, DRuby/Rubydust/rtc, Reticulated Python, and the various TypeScript dialects are all more or less in the dynamic-first lineage, since they are put on top existing dynamic languages. While Reticulated Python is inspired by gradual typing, the transient checking strategy they invented for it only loosely corresponds to anything found in the static-first lineage [67] (see Greenman and Felleisen [31] and Greenman and Migeed [32]).

There are noteworthy exceptions. C# is an example of a language with static types with added dynamic features; this effort doesn't seem particularly informed by gradual typing theory, but draws on some of the challenges here as motivation, e.g., working with JSON and XML.[9] Various recent systems have moved beyond core calculi, studying surface syntax directly [70, 39, 17, 18]; why not try practical experiments?

The best proof that I am wrong—that the distinction between these lineages is a trivial one and the theory is already applicable to practice—would be an interoperation layer for an existing statically typed language that follows existing theory directly, without any need to adapt the theory to pre-existing conditions. I would welcome such proof, and I encourage the gradual types community to take advantage of this opportunity and implement their ideas.

[9] `https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/dynamic-language-runtime-overview`

References

1. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 213–227, New York, NY, USA, 1989. ACM. URL: http://doi.acm.org/10.1145/75277.75296, doi:10.1145/75277.75296.
2. Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 201–214, 2011. URL: https://doi.org/10.1145/1926385.1926409, doi:10.1145/1926385.1926409.
3. Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for free for free: parametricity, with and without types. *PACMPL*, 1(ICFP):39:1–39:28, 2017. URL: https://doi.org/10.1145/3110283, doi:10.1145/3110283.
4. Beatrice Åkerblom, Jonathan Stendahl, Mattias Tumlin, and Tobias Wrigstad. Tracing dynamic features in python programs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 292–295, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2597073.2597103, doi:10.1145/2597073.2597103.
5. Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for smalltalk. *Sci. Comput. Program.*, 96(P1):52–69, December 2014. URL: http://dx.doi.org/10.1016/j.scico.2013.06.006, doi:10.1016/j.scico.2013.06.006.
6. Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. Confined gradual typing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 251–270, 2014. URL: https://doi.org/10.1145/2660193.2660222, doi:10.1145/2660193.2660222.
7. David An, Avik Chaudhuri, Jeffrey Foster, and Michael Hicks. Dynamic inference of static types for ruby. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)*, pages 459–472. ACM, 2011.
8. Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 283–295, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2628136.2628149, doi:10.1145/2628136.2628149.
9. Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strnisa, Jan Vitek, and Tobias Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 117–136, 2009. URL: https://doi.org/10.1145/1640089.1640098, doi:10.1145/1640089.1640098.
10. William J. Bowman and Amal Ahmed. Typed closure conversion for the calculus of constructions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 797–811, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3192366.3192372, doi:10.1145/3192366.3192372.
11. William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. Type-preserving cps translation of Σ and Π types is not not possible. *Proc. ACM Program. Lang.*, 2(POPL):22:1–22:33, December 2017. URL: http://doi.acm.org/10.1145/3158110, doi:10.1145/3158110.
12. John Boyland. The problem of structural type tests in a gradual-typed language. In *FOOL*, 2014.
13. Gilad Bracha. Pluggable type systems. In *In OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.
14. John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. Migrating gradual types. *Proc. ACM Program. Lang.*, 2(POPL):15:1–15:29, December 2017. URL: http://doi.acm.org/10.1145/3158103, doi:10.1145/3158103.

15 Robert Cartwright. User-defined data types as an aid to verifying lisp programs. In *ICALP*, 1976.

16 Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 278–292, New York, NY, USA, 1991. ACM. URL: http://doi.acm.org/10.1145/113445.113469, doi:10.1145/113445.113469.

17 Giuseppe Castagna and Victor Lanvin. Gradual typing with union and intersection types. *Proc. ACM Program. Lang.*, 1(ICFP):41:1–41:28, August 2017. URL: http://doi.acm.org/10.1145/3110285, doi:10.1145/3110285.

18 Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. Gradual typing: A new perspective. *Proc. ACM Program. Lang.*, 3(POPL):16:1–16:32, January 2019. URL: http://doi.acm.org/10.1145/3290329, doi:10.1145/3290329.

19 Matteo Cimini and Jeremy G. Siek. The gradualizer: a methodology and algorithm for generating gradual type systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 443–455, 2016. URL: https://doi.org/10.1145/2837614.2837632, doi:10.1145/2837614.2837632.

20 Tim Disney and Cormac Flanagan. Gradual information flow typing. In *STOP*, 2011.

21 Mike Fagan. *Soft typing: An approach to type checking for dynamically typed languages*. PhD thesis, 1991. URL: https://scholarship.rice.edu/handle/1911/16439.

22 L. Fennell and P. Thiemann. Gradual security typing with references. In *2013 IEEE 26th Computer Security Foundations Symposium*, pages 224–239, June 2013. doi:10.1109/CSF.2013.22.

23 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, pages 48–59, 2002. URL: https://doi.org/10.1145/581478.581484, doi:10.1145/581478.581484.

24 Cormac Flanagan. Hybrid type checking. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 245–256, 2006. URL: https://doi.org/10.1145/1111037.1111059, doi:10.1145/1111037.1111059.

25 Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael W. Hicks. Static type inference for ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*, pages 1859–1866, 2009. URL: https://doi.org/10.1145/1529282.1529700, doi:10.1145/1529282.1529700.

26 Ronald Garcia and Matteo Cimini. Principal type schemes for gradual programs. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 303–315, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2676726.2676992, doi:10.1145/2676726.2676992.

27 Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 429–442, 2016. URL: https://doi.org/10.1145/2837614.2837670, doi:10.1145/2837614.2837670.

28 Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 231–245, New York, NY, USA, 2005. ACM. URL: http://doi.acm.org/10.1145/1094811.1094830, doi:10.1145/1094811.1094830.

29 Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *POPL*, pages 353–364. ACM, 2010.

30 Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. *J. Funct. Program.*, 22(3):225–274, 2012.

31 Ben Greenman and Matthias Felleisen. A spectrum of type soundness and performance. *Proc. ACM Program. Lang.*, 2(ICFP):71:1–71:32, July 2018. URL: http://doi.acm.org/10.1145/3236766, doi:10.1145/3236766.

32 Ben Greenman and Zeina Migeed. On the cost of type-tag soundness. In *PEPM*, pages 30–39, 2018. doi:https://doi.org/10.1145/3162066.

33 Ben Greenmanand, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthia Felleisen. How to evaluate the performance of gradual type systems. *JFP*, 2018. Accepted and to appear.

34 Fritz Henglein. Dynamic typing: Syntax and proof theory. In *Selected Papers of the Symposium on Fourth European Symposium on Programming*, ESOP'92, pages 197–230, Amsterdam, The Netherlands, The Netherlands, 1994. Elsevier Science Publishers B. V. URL: http://dl.acm.org/citation.cfm?id=197475.190867.

35 Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *PACMPL*, 1(ICFP):38:1–38:28, 2017. URL: https://doi.org/10.1145/3110282, doi:10.1145/3110282.

36 Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. Efficient gradual typing. *CoRR*, abs/1802.06375, 2018. URL: http://arxiv.org/abs/1802.06375, arXiv:1802.06375.

37 Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing or, theorems for low, low prices! In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 16–31, 2008. URL: https://doi.org/10.1007/978-3-540-78739-6_2, doi:10.1007/978-3-540-78739-6_2.

38 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM Trans. Program. Lang. Syst.*, 31(3):12:1–12:44, 2009. URL: https://doi.org/10.1145/1498926.1498930, doi:10.1145/1498926.1498930.

39 Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. Dynamic type inference for gradual hindley–milner typing. *Proc. ACM Program. Lang.*, 3(POPL):18:1–18:29, January 2019. URL: http://doi.acm.org/10.1145/3290331, doi:10.1145/3290331.

40 Fabian Muehlboeck and Ross Tate. Sound gradual typing is nominally alive and well. In *OOPSLA*, New York, NY, USA, 2017. ACM. URL: http://www.cs.cornell.edu/~ross/publications/nomalive/, doi:https://doi.org/10.1145/3133880.

41 Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification for higher-order stateful programs. *PACMPL*, 2(POPL):51:1–51:30, 2018. URL: https://doi.org/10.1145/3158139, doi:10.1145/3158139.

42 Alberto Oliart. An algorithm for inferring quasi-static types. Technical Report 1994-013, Boston University, 1994. URL: http://www.cs.bu.edu/techreports/pdf/1994-013-quasi-static-types.pdf.

43 Daniel Patterson and Amal Ahmed. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:15, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7125, doi:10.4230/LIPIcs.SNAPL.2017.12.

44 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 167–180, 2015. URL: https://doi.org/10.1145/2676726.2676971, doi:10.1145/2676726.2676971.

45 Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. The ruby type checker. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing,*

SAC '13, Coimbra, Portugal, March 18-22, 2013, pages 1565–1572, 2013. URL: https://doi.org/10.1145/2480362.2480655, doi:10.1145/2480362.2480655.

46 Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do. In Mira Mezini, editor, *ECOOP 2011 – Object-Oriented Programming*, pages 52–78, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

47 Ilya Sergey and Dave Clarke. Gradual ownership types. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 579–599, Berlin, Heidelberg, 2012. Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-642-28869-2_29, doi:10.1007/978-3-642-28869-2_29.

48 Uri Shaked. Typewiz, 2019. URL: https://github.com/urish/typewiz.

49 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

50 Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, pages 2–27, 2007. URL: https://doi.org/10.1007/978-3-540-73589-2_2, doi:10.1007/978-3-540-73589-2_2.

51 Jeremy G. Siek and Sam Tobin-Hochstadt. The recursive union of some gradual types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 388–410, 2016. URL: https://doi.org/10.1007/978-3-319-30936-1_21, doi:10.1007/978-3-319-30936-1_21.

52 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined Criteria for Gradual Typing. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 274–293, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5031, doi:10.4230/LIPIcs.SNAPL.2015.274.

53 Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. Typing the numeric tower. In *Practical Aspects of Declarative Languages - 14th International Symposium, PADL 2012, Philadelphia, PA, USA, January 23-24, 2012. Proceedings*, pages 289–303, 2012. URL: https://doi.org/10.1007/978-3-642-27694-1_21, doi:10.1007/978-3-642-27694-1_21.

54 Nikhil Swamy, Michael Hicks, and Gavin M. Bierman. A theory of typed coercions and its applications. *SIGPLAN Not.*, 44(9):329–340, August 2009. URL: http://doi.acm.org/10.1145/1631687.1596598, doi:10.1145/1631687.1596598.

55 Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Position paper: Performance evaluation for gradual typing. In *STOP: Workshop on Script to Program Evolution*, New York, NY, USA, 2015. ACM. URL: http://www.ccs.neu.edu/home/types/publications/pe4gt/pe4gt.pdf.

56 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 793–810, 2012. URL: https://doi.org/10.1145/2384616.2384674, doi:10.1145/2384616.2384674.

57 Satish R. Thatte. Quasi-static typing. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 367–381, 1990. URL: https://doi.org/10.1145/96709.96747, doi:10.1145/96709.96747.

58 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006,*

Portland, Oregon, USA, pages 964–974, 2006. URL: https://doi.org/10.1145/1176617.1176755, doi:10.1145/1176617.1176755.

59 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 395–406, New York, NY, USA, 2008. ACM. URL: http://doi.acm.org/10.1145/1328438.1328486, doi:10.1145/1328438.1328486.

60 Sam Tobin-Hochstadt and Robert Bruce Findler. Cycles without pollution: A gradual typing poem. In *Proceedings for the 1st Workshop on Script to Program Evolution*, STOP '09, pages 47–57, New York, NY, USA, 2009. ACM. URL: http://doi.acm.org/10.1145/1570506.1570512, doi:10.1145/1570506.1570512.

61 Matías Toro, Ronald Garcia, and Éric Tanter. Type-driven gradual security with references. *ACM Trans. Program. Lang. Syst.*, 40(4):16:1–16:55, 2018. URL: https://dl.acm.org/citation.cfm?id=3229061.

62 Matías Toro, Ronald Garcia, and Éric Tanter. Type-driven gradual security with references. *ACM Trans. Program. Lang. Syst.*, 40(4):16:1–16:55, December 2018. URL: http://doi.acm.org/10.1145/3229061, doi:10.1145/3229061.

63 Matías Toro, Elizabeth Labrada, and Éric Tanter. Gradual parametricity, revisited. *PACMPL*, 3(POPL):17:1–17:30, 2019. URL: https://dl.acm.org/citation.cfm?id=3290330.

64 Matías Toro, Elizabeth Labrada, and Éric Tanter. Gradual parametricity, revisited. *Proc. ACM Program. Lang.*, 3(POPL):17:1–17:30, January 2019. URL: http://doi.acm.org/10.1145/3290330, doi:10.1145/3290330.

65 Matías Toro and Éric Tanter. A gradual interpretation of union types. In *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, pages 382–404, 2017. URL: https://doi.org/10.1007/978-3-319-66706-5_19, doi:10.1007/978-3-319-66706-5_19.

66 Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 45–56, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2661088.2661101, doi:10.1145/2661088.2661101.

67 Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big types in little runtime: Open-world soundness and collaborative blame for gradual type systems. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 762–774, New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3009837.3009849, doi:10.1145/3009837.3009849.

68 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 1–16, 2009. URL: https://doi.org/10.1007/978-3-642-00590-9_1, doi:10.1007/978-3-642-00590-9_1.

69 Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, pages 459–483, 2011. URL: https://doi.org/10.1007/978-3-642-22655-7_22, doi:10.1007/978-3-642-22655-7_22.

70 Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. Consistent subtyping for all. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 3–30, Cham, 2018. Springer International Publishing.