The Dynamic Practice and Static Theory of Gradual Typing

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, implementation-oriented 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? What progress has been made so far?

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

Keywords and phrases dynamic typing, gradual typing, static typing, implementation, theory, challenge problems

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.6

Acknowledgements I thank Sam Tobin-Hochstadt and David Van Horn for their hearty if dubious encouragment. Conversations with Sam, Ron Garcia, Matthias Felleisen, Robby Findler, and Spencer Florence improved the argumentation. Stephanie Weirich helped me with Haskell programming; Richard Eisenberg wrote the program in Figure 3. Ross Tate, Neel Krishnaswami, Ron, and Éric Tanter provided helpful references I had overlooked. Ben Greenman provided helpful feedback. The anonymous SNAPL reviewers provided insightful comments as well as the CDuce code in Section 3.4.3. Finally, I thank Stephen Chong and Harvard Computer Science for hosting me while on sabbatical.

3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 6; pp. 6:1–6:20

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 such that "code in typed modules can't go wrong" using *contracts* [102, 34]; 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* [87].[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] [101]. 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 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.

The differences at birth between these two approaches are still evident, and the latter approach has an opportunity for interesting new discoveries from proof-of-concept (and more serious) implementations.[2]

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" [117]. 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. [102]

---

[1] Flanagan showed how to use a similar cast framework to relax a fancy subset type system to a series of dynamic checks [36]. Dynamic checking is necessary in Flanagan's *hybrid typing*, because not every refinement is easy to send to an SMT solver. While the approach is different, the spirit is similar: there must have been something in the water.

[2] There are three other distinctions one could make. First, the macro/micro distinction from Takikawa et al. and Greenman et al. [98, 49]; second, the latent/manifest distinction [45, 46]; and third, the distinction between languages with static semantics that influence runtime behavior (e.g., type classes) and those languages where types can be erased. These distinctions are important but less salient for my analysis.

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 a canonical example, though TypeScript's various dialects, Dart, DRuby/Rubydust/rtc, Clojure's `specs`, Gradualtalk, and Reticulated Python are all comparable efforts in the research community [103, 39, 8, 81, 80, 5, 112]. These languages all share an approach going back chiefly to the 1990s but also earlier: we have a dynamic language in hand and we'd like the putative benefits of static typing (for, e.g., maintenance [59], documentation [64], performance) [20, 100, 17, 21, 50].

Dynamic idioms vary widely [82, 4], but a common theme is untypability in conventional systems. Accordingly, the type systems used in the dynamic-first approach tend to the unfamiliar, with features designed to capture particular language idioms: occurrence typing [103, 104, 56], "like" types [119], severe relaxations of runtime checking disciplines to avoid disrupting reference equality [112, 113], and ad hoc rules for inferring particular types (e.g., telling the difference between a tuple and an array in TypeScript or Flow).

1.2 The static-first approach

Siek and Taha take a "micro" approach, where the unit of interoperation is the expression [87]. They are inspired by Thatte's quasi-static typing and Oliart's algorithmic treatment thereof [100, 73]. 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.

Siek and Taha's paper does not, however, identify any particular dynamic idioms they want to write but 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. [99]) or to write new code that goes beyond their system (e.g., as in Tobin-Hochstadt and Findler [106]). To be sure, adding the dynamic type does add a new behavior – nontermination [1]. Siek and Taha don't explicitly observe as much beyond mentioning that the untyped lambda calculus embeds in their system. The code of their two lambda calculus interpreters is identical (their Figure 1; reproduced in our Figure 1); only the type annotations change.

According to Siek et al.'s refined definition [90], 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 [41]).

Work in the "static-first" lineage cites interoperation as a motivation, not only in Siek and Taha's seminal paper [87] but especially in Wadler and Findler [114]. Later papers take interesting type feature X and show how to relax the typing rules, resolving static imprecision with dynamic checks. X ranges widely: objects [88], polymorphism [2, 3, 108], typestate [118],

```
(define interp
  (λ (env e)
    (case e
      [(Var ,x) (cdr (assq x env))]
      [(Int ,n) n]
      [(App ,f ,arg) (apply (interp env f) (interp env arg))]
      [(Lam ,x ,e) (list x e env)]
      [(Succ ,e) (succ (interp env e))])))

(define apply
  (λ (f arg)
    (case f
      [(,x ,body ,env)
       (interp (cons (cons x arg) env) body)]
      [,other (error "in application, expected a closure")])))
```

```
(type expr (datatype (Var ,symbol)
                     (Int ,int)
                     (App ,expr ,expr)
                     (Lam ,symbol ,expr)
                     (Succ ,expr)))
(type envty (listof (pair symbol ?)))

(define interp
  (λ ((env : envty) (e : expr))
    (case e
      [(Var ,x) (cdr (assq x env))]
      [(Int ,n) n]
      [(App ,f ,arg) (apply (interp env f) (interp env arg))]
      [(Lam ,x ,e) (list x e env)]
      [(Succ ,e) (succ (interp env e))])))

(define apply
  (λ (f arg)
    (case f
      [(,x ,body ,env)
       (interp (cons (cons x arg) env) body)]
      [,other (error "in application, expected a closure")])))
```

Figure 1 The lambda calculus interpreter from Siek and Taha (Figure 1 [87]).

information flow control [29, 33, 107], ownership types [85], effects [9], session types [55], etc. The process of relaxation was characterized and made beautifully concrete in Garcia, Clark, and Tanter's "Abstracting Gradual Typing" (AGT) [42]. In AGT, one "abstracts" a gradual type system starting from a syntax-directed, fully static type system that enjoys a preservation-based proof of type safety. Matteo Cimini and Jeremy Siek built the Gradualizer, a tool for automatically turning a variety of type systems gradual [27]. AGT is a human methodology, but has been shown to apply to a broad swath of systems [9, 61, 107, 108]. The Gradualizer is automatic, but is substantially less general than the principles in AGT.

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, where statically typed programs remain acceptable – satisfying the static gradual guarantee (removing type information retains typeability) [90]. Many systems also enjoy the dynamic gradual guarantee (removing type information retains successful runs of the program), though notably not for several type systems implementing hyperproperties [107, 108].

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 [103]

2.1 A distinction without a difference?

Does it matter whether one starts from dynamic typing and works up to static typing or starts with static typing and relaxes to allow dynamic typing [53]? Only the dynamic-first lineage addresses particular examples and the particular difficulties they introduce into the resulting systems.

```
1 (define (flatten x)
2   (cond
3     [(null? x) '()]
4     [(cons? x) (append (flatten (car x)) (flatten (cdr x)))]
5     [else      (list x)]))
```

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

Figure 2 The `flatten` function in Scheme/Racket.

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`. [8]

Static-first gradual typing typically lacks such concrete examples as motivation, studying interoperation more abstractly. Static-first gradual typing often studies type system *features* without any attempt to accommodate the idiosyncrasies of any particular implementation of those features. (There are, of course, laudable exceptions [85, 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 [103, 39, 8, 81, 80, 5, 112]. There are several notable exceptions: Nom and Grift are direct implementations of the static-first theory of gradual typing for new static languages [68, 60]; Thorn invents its own theory of "like" types [13]; C# is a statically typed language which grew a dynamic runtime unrelated to the theory of gradual types [65].

It is surprising that the theory takes 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 idioms from dynamic typing 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 2). 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 ? → `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 uniformly constructed, heterogeneous lists cannot be written down in simple type languages. Like in Tobin-Hochstadt and Findler's "gradual typing poem" [106], 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).

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.[3]

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 by its lack of adoption 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 suffices to characterize many of the idioms used: it "accommodates ... modes of reasoning ... programmers use" – Typed Racket was designed "to support Scheme idioms and programming styles" [103]. To put it in terms of Ron Garcia's 2018 ICFP keynote, Typed Racket is an exercise in "type system anthropology", finding the folk type system that corresponds to Racket programmers' mental models [40].

2.2.2 flatten in static-first gradual typing

How might one write `flatten` in the static-first lineage? First, let us be clear that statically typed languages can already more or less accommodate the `flatten` function! Zhe Yang implemented it in SML two ways: once with functors, and once with embeddings to and projections from a universal-datatype [121]; the embedding/projection model is not too hard to use but is not the most efficient [10, 11]. One can implement `flatten` in Haskell using recent reflection support (see Figure 3). ℂDuce can express this function directly (see Figure 4).

Static languages accommodate `flatten` with either significant runtime cost or fancy type systems. Work in the static-first lineage of gradual typing has only recently devised systems that can accommodate this simple function. Most static-first gradual type systems don't offer type tests, though there are noteworthy exceptions [62, 63, 16]. Siek and Tobin-Hochstadt's true union types [89] can handle the definition at the same moral type of ? → `list` ? (in their notation, $\star \to \mu X.\texttt{unit} \cup \star \times X$). Recent work by Castagna, Lanvin, and others might be able to accommodate the idiom, as well [24, 25].

3 An opportunity

We ought 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, leading to refinements and improvements in both theory and practice.

Since there are multiple motivations for wanting gradual typing, I've broken the challenges up into sections by motivation: added expressiveness (Section 3.1), interoperation (Section 3.2), and types themselves (Section 3.3). I conclude by addressing possible objections (Section 3.4).

[3] Typed Racket assigns the type `(-> Any (Listof Any))`. Unfortunately, Typed Racket cannot express the negation lurking in the codomain under the `Listof`, where one might want to write `(-> Any (Listof (- Any (Listof Any))))`.

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.
>
> – Mike Fagan's *Fundamental Theorem of Static Typing* [32]

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: heterogeneous structures, semi-structured data, and an object annotation strategy drawn from the "middleware" approach to webservers.

3.1.1 Heterogeneous structures

It is very common to program with uniformly constructed, heterogeneous data structures in dynamic programming languages: the lists in `flatten` nest arbitrarily and hold arbitrary values (heterogeneity) but are constructed using only `'()` and `cons` (uniformity). While `flatten` is a "toy" function, it manipulates the heterogeneous lists with a non-trivial use of type predicates in a way that is simultaneously realistic but also challenging to existing static-first type theories. Fagan's PhD thesis is rich in such examples [32].

Not only do static type systems limit the kinds of values that get put into data structures, they often limit the shapes of those data structures themselves. It is not a trivial exercise to construct a non-statically known, immutable, circular list in OCaml. Programming in a static language, I might want to temporarily "cheat" and view my structured data a little less formally than the type system would ordinarily allow. For example, one might temporarily allow mutation to make a list circular before "freezing" it as an immutable one [106]. How can such shenanigans be safely accommodated in languages that want types to mean things? What does heterogeneity mean for more complicated structures like tree-based sets and maps that, e.g., compare values to maintain invariants?

3.1.2 Semi-structured data, like JSON, YAML, and XML

A great deal of information is stored and exchanged in semi-structured formats 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. Much of XML can be handled with some moderately fancy type system features – unions and recursive types [12] – but a proper account of names has proven elusive, in part due to the challenging type system features necessary (e.g., row types and first-class names). While Typed Racket is good at working with heterogeneous structures, it accommodates semi-structured data less well. Can gradual typing help us work with these common structures? Might we be able to gradualize recent advances in reasoning about rows [67]? Might we offer a gradual treatment of the row-based metaprogramming of Ur/Web [26]?

3.1.3 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. Middleware can be used for many tasks: application transformers which, e.g., compress outgoing data with gzip, but also session management and authentication regimes for tracking which user a request belongs to. Such authentication middleware might look up user information and then *attach* that user information to the request object, making it available for other portions of the web application that rely on such user information being present.

Authentication middleware amounts to a form of strong update, where a record – the HTTP request object – gets a new field, or its field changes type. Sound gradual systems can support strong update [91]; can we extend and implement these systems to write *mostly* typed web-servers that can accommodate this "attachment" idiom?

3.2 Gradual typing for interoperation

If one studies gradual typing in order 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. High level concerns include design issues surrounding numerics, annotations, type-driven features, and how data structures (and their invariants) can be ported from one language to another. There are also critical lower level concerns, like garbage collection, linking, and debugging.

3.2.1 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` and other conversions in Haskell's numeric type classes) 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 [93]; what should happen in other settings?

3.2.2 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" [102]. 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,[4] 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 translate it to one of many possible targets in B. Such translations will come with a computational 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. Can gradual typing theory or implementations help us think about these questions? There has been some related work for contracts and data structures [35]; how might it port over?

3.2.3 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 [68]. The core issue is that the runtime semantics of some features depend on the decisions made during static typing. Haskell's type classes are

[4] Though Racket documentation indicates that not all strings are immutable [77].

another example of this phenomenon: it is determined statically how to resolve each call of a constrained function, which determines, e.g., which monad to use. 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?

3.2.4 Minimizing annotation overhead

Static-first gradual typing typically studies elaborated core calculi – many papers do not describe the surface language that generates the runtime checks. How can we minimize the annotation overhead in the source language? What check insertion strategies are appropriate? Swamy et al. give a theoretical starting point for thinking about coercions more generally [96], subsuming Henglein's seminal work [50] but not offering an algorithm.. Allende et al. offer a concrete analysis of the issue and a novel, hybrid cast insertion strategy in an object-oriented setting [7]. Greenman and Felleisen consider "a spectrum of type soundness" for cast insertion but not alternative sound strategies [47]; What tool support do we need – inference [86]? Something more exploratory, along the lines of Campora et al. [18, 19]? More tools for eliminating checks [71]?

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 [29, 33, 107]). Experiments with an implementation are a natural next step.

3.2.5 Lower-level concerns: garbage collection, linking, and debugging

When two languages interoperate, which is responsible for allocating and deallocating? When does each language's GC run? This question is a serious one: in Ramsey's Lua-ML, the thorny issue of whose garbage collector is in charge led him to reimplement Lua in OCaml [79]! Not only is such a "duplication of effort ... regrettable", it means that Ramsey's Lua may not behave identically to the original Lua and won't necessarily keep up as Lua evolves.

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! How does one take the hodgepodge of stack frames, thunks, and continuations from mixing two real languages and produce something intelligible?

3.3 Gradual typing for typing's sake

One could summarize the gradual types approach as finding runtime-enforceable safety properties that simultaneously (a) allow one to relax the strictures of type checking in part of one's program while (b) not compromising the safety guarantees in the checked parts of the program. But types are more than safety guarantees! Folklore and substantial engineer experience assign high value to what I called before the "putative benefits of static typing". Types are executable documentation [64]; they rule out whole classes of errors, assist in maintenance [59], and can lead to more efficient code. So far, the literature on gradual types has focused on the "rule out whole classes of errors" benefit. But there are others! To what extent do the existing, implemented dynamic-first systems buy you the benefits of static typing? To what extent would implementations of theoretical, static-first systems buy you those same benefits?

For efficiency, it's a mixed bag. Typed Racket sometimes generates better code than one would get without the type systems [78]. But Typed Racket has a hard row to hoe: it lives within the strictures of Racket's dynamic-first world, where the primitives by default perform

runtime checks. Typed Racket already adds its own checks at module boundaries; avoiding checks on internal uses of primitives takes both effort and care. Working in ActionScript, which has a less constrained runtime, Rastogi et al. find an average 1.6x improvement when using their type inference algorithm over partially annotated programs. Nom and Grift both show that gradual types can be implemented efficiently [68, 60]; while both can generate good static code, neither recovers global, type-based optimizations.

For maintenance, Typed Racket exhibits some of the desirable behaviors: on changing an algebraic datatype definition to include new possibilities, the type system can find functions that are now missing cases. But having the right features in the small and behaving correctly in the large are two different things. A ten-year retrospective on Typed Racket's "migratory typing" suggests that deeper study is required [105].

3.4 In which I am gravely mistaken

"No, no," you say, "that's not right. We can already do all of this!" There has indeed been good progress towards meeting these challenges! I don't mean to diminish the substantial literature on these topics, but rather to help direct its aims. What do we have so far, and how close are we to meeting the challenges I've laid out?

3.4.1 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. Polymorphic variants offer some of the flexibility and reuse of dynamic types while also admitting a meaningful typing discipline [43].

Standard examples like heterogeneous lists and maps are typeable using some of the fancier features of Haskell's type system [58, 57, 115]. 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.

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, even if some claim (tongue in cheek) that Haskell is already gradually typed [31]. For example, the `flatten` function can be written in Haskell (Figure 3), but it is somewhat less readable than its Racket counterpart. The definition itself is not so much longer than the Racket code, and the supporting functions could live in a library. One could presumably write a similar program in Scala using the `Dynamic` trait.

Or consider Yesod, a mature Haskell web application framework [92]. Yesod uses idioms like routing and middleware for specifying servers, as is common in dynamic web frameworks like ExpressJS [37].

Yesod supports sessions directly, using cookies to allow a notion of continuity in stateless HTTP sessions. Yesod's sessions are implemented as maps from `Text` to `ByteString`. Could one build a version of Yesod that used Haskell's type system to guarantee that user information was available in the session map for stages of processing that needed it? Could we construct a gradual version of that system?

3.4.2 We can use linking types

Patterson and Ahmed's linking types solve this problem [76].

▶ Response. Let's implement it! Linking types have been successful for proving things about translations [15, 14]. Do they have any bearing on implementations? Work by Matthews et al. offers some gestures in this direction [63, 62];

```haskell
{-# LANGUAGE
    GADTs, TypeApplications, ScopedTypeVariables, ViewPatterns,
    PolyKinds, DataKinds
#-}
module Flatten where

import Data.Dynamic
import Type.Reflection

data MaybeMatch (a :: k1) (b :: k2) where
  Match :: MaybeMatch a a
  NoMatch :: MaybeMatch a b

isType :: forall a b. Typeable a => TypeRep b -> MaybeMatch a b
isType (eqTypeRep (typeRep @a) -> Just HRefl) = Match
isType _ = NoMatch

smartToDyn :: TypeRep a -> a -> Dynamic
smartToDyn (isType @Dynamic -> Match) x = x
smartToDyn rep                        x = Dynamic rep x

flatten :: [Dynamic] -> [Dynamic]
flatten [] = []
flatten (dx@(Dynamic rep x):dxs) = x' ++ flatten dxs
  where
    x' | App (isType @[] -> Match) arg <- rep
       = flatten (map (smartToDyn arg) x)
       | otherwise
       = [dx]
```

Figure 3 A version of `flatten` using Haskell's `Dynamic` type.

3.4.3 These ideas are already implemented

Typed Racket, Gradualtalk, DRuby/rtc/Rubydust, Reticulated Python, Thorn, Nom and Grift are implementations [103, 5, 39, 8, 81, 112, 13, 68, 60]; some theoretical work offers web interfaces for experimentation with their type theory [107].

▶ 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 of 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 [113] (see Greenman and Felleisen [47] and Greenman and Migeed [48]).

Of course, there are exceptions. C# is an example of a statically typed language that added dynamic features; this effort doesn't seem particularly informed by gradual typing theory, but its dynamic language runtime draws on some of the challenges here as motivation, e.g., working with JSON and XML [65].

Nick Benton showed how to use `call/cc` to get clean errors at the dynamic/static interface when mixing untyped and typed code [11], but never scaled up to a real language. Similar embedding/projection pairs can be found in Yang's work, Benton's earlier work on embedding languages in ML, and Ramsey's Lua-ML [121, 10, 79]. The idea of embedding/projection

```
let flatten ( Any -> [ (Any\[Any*])* ] )
  | []     -> []
  | (h,t) -> (flatten h)@(flatten t)
  | x      -> [x]

type Tree('a) = ('a\[Any*]) | [ (Tree('a))* ]

let flattenTree ( (Tree('a)) -> ['a*] )
   | []       -> []
   | [h ; t] -> (flattenTree h)@(flattenTree t)
   | x        -> [x]
```

Figure 4 Flatten in ℂDuce, with and without a custom type.

pairs as a core idea in gradual types goes back at least to Henglein [51], but has seen a resurgence in recent work by New et al. [69, 70]. Only Ramsey's work seems to have ever enjoyed a serious implementation; even so, it seems that the up-to-date Lua-OCaml interface is via a more conventional, `ctypes`-based API [30]. Gray et al.'s reflection-based approach offer a substantial interface between two very different languages (Java and Scheme) [44]. Their interface compiles the Java to Scheme, though, sidestepping the "two runtimes" issue in much the same way Ramsey does.

ℂDuce deserves particular attention, as its set-theoretic types allow for concise implementations of functions like `flatten` [12]. We can write two good implementations of `flatten` in ℂDuce (Figure 4): the first version uses ordinary types, while the second uses a custom type definition to better characterize the list structure of the input. ℂDuce doesn't address issues of interoperation at all, but recent work has shown that ℂDuce's set-theoretic types are relevant to gradual typing [25]. How much of the reasoning done in dynamic languages can be done using set-theoretic types? What kinds of reasoning lie outside those types?

Various recent systems have moved beyond core calculi, studying surface syntax directly [120, 66, 24, 25]; why not try practical experiments?

3.4.4 That's not what gradual typing is about

The dynamic and static ends of the gradual typing spectrum are relative. As Ron Garcia shows in his 2018 ICFP keynote, one could consider SML as a dynamic language in light of the static verification of partial pattern matches used in the datasort refinement system of Refined ML [38, 40]. Some of the gradual notions of, e.g., security typing [29, 33, 107], have this flavor.

▶ Response. The particular challenge problems may change, but a focus on implementations will help the community relate efforts in the world of gradual types and efforts outside.

Taking security typing as a concrete instance, there are already researchers working hard on runtime enforcement mechanisms for information flow control [95, 54, 52, 94, 110]. While these systems generally don't support a notion of graduality, they *do* support important security properties overlooked in the gradual types approaches (for example, protection from side channels and termination sensitivity). It would be productive to hold a "Build It, Break It, Fix it" contest between gradual and conventional systems [83]. LIO's implementation has already been used to build systems of moderate size [75]; why not build a comparable system in a gradual security-typed language?

There has been some work on enforcing linearity at runtime, with ALMS being a realistic implementation [109]; Scherer et al.'s theory comes with a toy implementation [84]. To test these ideas, why not implement such a system for Rust, allowing runtime checking for affinity as an alternative to unsafe blocks?

Finally, the first question asked after Ron Garcia's keynote was, "I'd like to use 'dynamic' Haskell programs from Agda. What do I do about nontermination?" A good deal of work studies how to gracefully allow nontermination in languages with consistent logical interpretations of their types [116, 22, 74, 23, 97, 111, 28]. What might a "gradual" system look like here? Does the AGT methodology help?

The gradual types approach is to find safety properties – i.e., runtime enforcement mechanisms – that are sufficient to guarantee the desirable properties of a given type system. Nguyen et al.'s runtime semantics for checking size-change termination is related: their runtime checks can be lifted to static ones via an analysis [72]. Their implementation isn't "gradual", though, as there's no notion of "mixed" dynamic and static checking.

More broadly, is "graduality" even the right fit for thinking about nontermination? There is some recent evidence that the dynamic gradual guarantee – which some see as essential to gradual typing [70] – is incompatible with various hyperproperties, like noninterference [107] and parametricity [108]. Binary formulations of parametricity generalize unary formulations of termination arguments. Can a language satisfy the gradual guarantees but also preserve strong properties like logical consistency in mixed programs?

4 Conclusion

I come to praise gradual types, not to bury them.

– with apologies to William Shakespeare

Gradual types is a thriving research area. Work is plentiful in both the implementation-focused dynamic-first lineage and the theoretically-focused static-first lineage. Our community has made good progress so far, and recent implementation efforts give me hope that the two lineages will enter into a more fruitful dialog. Significant challenges remain for both the theory and practice of gradual types; there are thorny practical questions that deserve immediate attention, as they will determine the direction of our efforts. Which programs do we want to write? How do theoretical models generalize to whole languages? What of the low-level concerns of interoperation, viz. the "two runtimes" problem?

Finally, perhaps I am wrong. Maybe the distinction between these lineages is a trivial one, and the theory is already applicable to practice. The best proof that the distinction I draw is trivial would be an interoperation layer for an existing language that follows existing theory directly, without any need to adapt 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.

References

1 M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic Typing in a Statically-typed Language. In *Principles of Programming Languages (POPL)*, pages 213–227, New York, NY, USA, 1989. ACM. doi:10.1145/75277.75296.

2 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *Principles of Programming Languages (POPL)*, pages 201–214, 2011. 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. doi:10.1145/3110283.

4 Beatrice Åkerblom, Jonathan Stendahl, Mattias Tumlin, and Tobias Wrigstad. Tracing Dynamic Features in Python Programs. In *Working Conference on Mining Software Repositories (MSR)*, pages 292–295, New York, NY, USA, 2014. ACM. 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. doi:10.1016/j.scico.2013.06.006.

6 Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. Confined gradual typing. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 251–270, 2014. doi:10.1145/2660193.2660222.

7 Esteban Allende, Johan Fabry, and Éric Tanter. Cast Insertion Strategies for Gradually-typed Objects. In *Dynamic Languages Symposium (DLS)*, pages 27–36, New York, NY, USA, 2013. ACM. doi:10.1145/2508168.2508171.

8 David An, Avik Chaudhuri, Jeffrey Foster, and Michael Hicks. Dynamic Inference of Static Types for Ruby. In *Principles of Programming Languages (POPL)*, pages 459–472. ACM, 2011.

9 Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. A Theory of Gradual Effect Systems. In *International Conference on Functional Programming (ICFP)*, pages 283–295, New York, NY, USA, 2014. ACM. doi:10.1145/2628136.2628149.

10 Nick Benton. Embedded Interpreters. *J. Funct. Program.*, 15(4):503–542, July 2005. doi:10.1017/S0956796804005398.

11 Nick Benton. Undoing Dynamic Typing (Declarative Pearl). In Jacques Garrigue and Manuel V. Hermenegildo, editors, *Functional and Logic Programming*, pages 224–238, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

12 Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In *International Conference on Functional Programming (ICFP)*, pages 51–63. ACM, 2003. doi:10.1145/944705.944711.

13 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 *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 117–136, 2009. doi:10.1145/1640089.1640098.

14 William J. Bowman and Amal Ahmed. Typed Closure Conversion for the Calculus of Constructions. In *Programming Language Design and Implementation (PLDI)*, pages 797–811, New York, NY, USA, 2018. ACM. doi:10.1145/3192366.3192372.

15 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. doi:10.1145/3158110.

16 John Boyland. The Problem of Structural Type Tests in a Gradual-Typed Language. In *FOOL*, 2014.

17 Gilad Bracha. Pluggable type systems, October 2004. URL: http://bracha.org/pluggableTypesPosition.pdf.

18 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. doi:10.1145/3158103.

19 John Peter Campora, Sheng Chen, and Eric Walkingshaw. Casts and Costs: Harmonizing Safety and Performance in Gradual Typing. *Proc. ACM Program. Lang.*, 2(ICFP):98:1–98:30, July 2018. doi:10.1145/3236793.

20 Robert Cartwright. User-Defined Data Types as an Aid to Verifying LISP Programs. In *ICALP*, 1976.

21 Robert Cartwright and Mike Fagan. Soft Typing. In *Programming Language Design and Implementation (PLDI)*, pages 278–292, New York, NY, USA, 1991. ACM. doi:10.1145/113445.113469.

22 Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Step-Indexed Normalization for a Language with General Recursion. In *MSFP*, volume 76, pages 25–39, 2012.

23 Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. In *Principles of Programming Languages (POPL)*, pages 33–46. ACM, 2014.

24 Giuseppe Castagna and Victor Lanvin. Gradual Typing with Union and Intersection Types. *Proc. ACM Program. Lang.*, 1(ICFP):41:1–41:28, August 2017. doi:10.1145/3110285.

25 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. doi:10.1145/3290329.

26 Adam Chlipala. Ur: Statically-typed Metaprogramming with Type-level Record Computation. In *Programming Language Design and Implementation (PLDI)*, pages 122–133, New York, NY, USA, 2010. ACM. doi:10.1145/1806596.1806612.

27 Matteo Cimini and Jeremy G. Siek. The gradualizer: a methodology and algorithm for generating gradual type systems. In *Principles of Programming Languages (POPL)*, pages 443–455, 2016. doi:10.1145/2837614.2837632.

28 Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. Foundations of dependent interoperability. *Journal of Functional Programming*, 28:e9, 2018. doi:10.1017/S0956796818000011.

29 Tim Disney and Cormac Flanagan. Gradual Information Flow Typing. In *Workshop on Script-to-Program Evolution (STOP)*, 2011.

30 Paolo Donadeo. Lua-OCaml. URL: https://pdonadeo.github.io/ocaml-lua/.

31 Richard Eisenberg. Haskell as a gradually typed dynamic language, January 2016. URL: https://typesandkinds.wordpress.com/2016/01/22/haskell-as-a-gradually-typed-dynamic-language/.

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

33 L. Fennell and P. Thiemann. Gradual Security Typing with References. In *Computer Security Foundations Symposium (CSF)*, pages 224–239, June 2013. doi:10.1109/CSF.2013.22.

34 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, pages 48–59, 2002. doi:10.1145/581478.581484.

35 Robert Bruce Findler, Shu-Yu Guo, and Anne Rogers. Lazy Contract Checking for Immutable Data Structures. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *Implementation and Application of Functional Languages*, pages 111–128. Springer-Verlag, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-85373-2_7.

36 Cormac Flanagan. Hybrid type checking. In *Principles of Programming Languages (POPL)*, pages 245–256, 2006. doi:10.1145/1111037.1111059.

37 The Node Foundation. ExpressJS. URL: https://expressjs.com/.

38 Tim Freeman and Frank Pfenning. Refinement types for ML. In *Programming Language Design and Implementation (PLDI)*, June 1991.

39 Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael W. Hicks. Static type inference for Ruby. In *Symposium on Applied Computing (SAC)*, pages 1859–1866, 2009. doi:10.1145/1529282.1529700.

40 Ronald Garcia. Gradual Typing (keynote), 2018. ICFP. URL: https://www.youtube.com/watch?v=fQRRxaWsuxI.

41 Ronald Garcia and Matteo Cimini. Principal Type Schemes for Gradual Programs. In *Principles of Programming Languages (POPL)*, pages 303–315, New York, NY, USA, 2015. ACM. doi:10.1145/2676726.2676992.

42 Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Principles of Programming Languages (POPL)*, pages 429–442, 2016. doi:10.1145/2837614.2837670.

43 Jacques Garrigue. Code reuse through polymorphic variants. In *FOSE*, 2000.

44 Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained Interoperability Through Mirrors and Contracts. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 231–245, New York, NY, USA, 2005. ACM. doi:10.1145/1094811.1094830.

45 Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Principles of Programming Languages (POPL)*, pages 353–364. ACM, 2010.

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

47 Ben Greenman and Matthias Felleisen. A Spectrum of Type Soundness and Performance. *Proc. ACM Program. Lang.*, 2(ICFP):71:1–71:32, July 2018. doi:10.1145/3236766.

48 Ben Greenman and Zeina Migeed. On the Cost of Type-Tag Soundness. In *Partial Evaluation and Program Manipulation PEPM*, pages 30–39, 2018. doi:10.1145/3162066.

49 Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. How to evaluate the performance of gradual type systems. *Journal of Functional Programming*, 29:e4, 2019. doi:10.1017/S0956796818000217.

50 Fritz Henglein. Dynamic Typing: Syntax and Proof Theory. In *European Symposium on Programming (ESOP)*, pages 197–230, Amsterdam, The Netherlands, The Netherlands, 1994. Elsevier Science Publishers B. V. URL: http://dl.acm.org/citation.cfm?id=197475.190867.

51 Fritz Henglein. Dynamic Typing: Syntax and Proof Theory. *Sci. Comput. Program.*, 22(3):197–230, June 1994. doi:10.1016/0167-6423(94)00004-2.

52 Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. IFC inside: Retrofitting languages with dynamic information flow control. In *POST*, volume 9036, pages 11–31. Springer, 2015.

53 David Van Horn, September 2018. URL: https://twitter.com/lambda_calculus/status/1039702266679369730.

54 Cătălin Hriţcu, Michael Greenberg, Ben Karel, Benjamin C. Pierce, and Greg Morrisett. All Your IFCException Are Belong to Us. In *Security and Privacy (SP)*, pages 3–17, 2013. I am deeply embarrassed by the title of this paper. doi:10.1109/SP.2013.10.

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

56 Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. Occurrence typing modulo theories. In *Programming Language Design and Implementation (PLDI)*, pages 296–309. ACM, 2016.

57 Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. *CoRR*, abs/cs/0509027, 2005.

58 Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly Typed Heterogeneous Collections. In *ACM SIGPLAN Workshop on Haskell*, pages 96–107, New York, NY, USA, 2004. ACM. doi:10.1145/1017472.1017488.

59 S. Kleinschmager, R. Robbes, A. Stefik, S. Hanenberg, and E. Tanter. Do static type systems improve the maintainability of software systems? An empirical study. In *IEEE International Conference on Program Comprehension (ICPC)*, pages 153–162, June 2012. doi:10.1109/ICPC.2012.6240483.

60 Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. Efficient Gradual Typing. In *Programming Language Design and Implementation (PLDI)*, 2019. To appear. arXiv:1802.06375.

61 Nico Lehmann and Éric Tanter. Gradual Refinement Types. In *Principles of Programming Languages (POPL)*, pages 775–788, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009856.

62 Jacob Matthews and Amal Ahmed. Parametric Polymorphism through Run-Time Sealing or, Theorems for Low, Low Prices! In *European Symposium on Programming (ESOP)*, pages 16–31, 2008. doi:10.1007/978-3-540-78739-6_2.

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

64 Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An Empirical Study of the Influence of Static Type Systems on the Usability of Undocumented Software. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 683–702, New York, NY, USA, 2012. ACM. doi:10.1145/2384616.2384666.

65 Microsoft. Dynamic Language Runtime overview. URL: https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/dynamic-language-runtime-overview.

66 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. doi:10.1145/3290331.

67 J. Garrett Morris and James McKinna. Abstracting Extensible Data Types: Or, Rows by Any Other Name. *Proc. ACM Program. Lang.*, 3(POPL):12:1–12:28, January 2019. doi:10.1145/3290325.

68 Fabian Muehlboeck and Ross Tate. Sound Gradual Typing is Nominally Alive and Well. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, New York, NY, USA, 2017. ACM. doi:10.1145/3133880.

69 Max S. New and Amal Ahmed. Graduality from embedding-projection pairs. *PACMPL*, 2(ICFP):73:1–73:30, 2018.

70 Max S. New, Daniel R. Licata, and Amal Ahmed. Gradual type theory. *PACMPL*, 3(POPL):15:1–15:31, 2019.

71 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. doi:10.1145/3158139.

72 Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Size-Change Termination as a Contract. In *Programming Language Design and Implementation (PLDI)*, 2019. To appear. arXiv:1808.02101.

73 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.

74 Peter-Michael Osera, Vilhelm Sjöberg, and Steve Zdancewic. Dependent interoperability. In *PLPV*, pages 3–14. ACM, 2012.

75 James Parker, Niki Vazou, and Michael Hicks. LWeb: Information Flow Security for Multi-tier Web Applications. *Proc. ACM Program. Lang.*, 3(POPL):75:1–75:30, January 2019. doi:10.1145/3290388.

76 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, *Summit on Advances in Programming Languages (SNAPL)*, volume 71, pages 12:1–12:15, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.SNAPL.2017.12.

77 PLT. URL: https://docs.racket-lang.org/reference/strings.html.

78 PLT. URL: https://docs.racket-lang.org/ts-guide/optimization.html.

79 Norman Ramsey. Embedding an Interpreted Language Using Higher-order Functions and Types. In *Workshop on Interpreters, Virtual Machines and Emulators (IVME)*, pages 6–14, New York, NY, USA, 2003. ACM. doi:10.1145/858570.858571.

80 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In *Principles of Programming Languages (POPL)*, pages 167–180, 2015. doi:10.1145/2676726.2676971.

81 Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. The ruby type checker. In *Symposium on Applied Computing (SAC)*, pages 1565–1572, 2013. doi:10.1145/2480362.2480655.

82 Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The Eval That Men Do. In Mira Mezini, editor, *European Conference on Object-Oriented Programming (ECOOP)*, pages 52–78, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

83 Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. Mazurek, and Piotr Mardziel. Build It, Break It, Fix It: Contesting Secure Development. In *Computer and Communications Security (CCS)*, pages 690–703, New York, NY, USA, 2016. ACM. doi:10.1145/2976749.2978382.

84 Gabriel Scherer, Max New, Nick Rioux, and Amal Ahmed. Fabous Interoperability for ML and a Linear Language. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 146–162, Cham, 2018. Springer International Publishing.

85 Ilya Sergey and Dave Clarke. Gradual Ownership Types. In *European Symposium on Programming (ESOP)*, pages 579–599, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-28869-2_29.

86 Uri Shaked. TypeWiz, 2019. URL: https://github.com/urish/typewiz.

87 Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

88 Jeremy G. Siek and Walid Taha. Gradual Typing for Objects. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 2–27, 2007. doi:10.1007/978-3-540-73589-2_2.

89 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. doi:10.1007/978-3-319-30936-1_21.

90 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, *Summit on Advances in Programming Languages (SNAPL)*, volume 32, pages 274–293, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.SNAPL.2015.274.

91 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic References for Efficient Gradual Typing. In Jan Vitek, editor, *European Symposium on Programming (ESOP)*, pages 432–456, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

92 Michael Snoyman et al. Yesod. URL: https://www.yesodweb.com/.

93 Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. Typing the Numeric Tower. In *Practical Aspects of Declarative Languages (PADL)*, pages 289–303, 2012. doi:10.1007/978-3-642-27694-1_21.

94 Deian Stefan, David Mazières, John C. Mitchell, and Alejandro Russo. Flexible dynamic information flow control in the presence of exceptions. *J. Funct. Program.*, 27:e5, 2017.

95 Deian Stefan, Alejandro Russo, John Mitchell, and David Maziéres. Flexible Dynamic Information Flow Control in Haskell. In *Haskell Symposium*, 2011.

96 Nikhil Swamy, Michael Hicks, and Gavin M. Bierman. A Theory of Typed Coercions and Its Applications. In *International Conference on Functional Programming (ICFP)*, pages 329–340, New York, NY, USA, 2009. ACM. doi:10.1145/1596550.1596598.

97 Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent Types and Multi-monadic Effects in F*. In *Principles of Programming Languages (POPL)*, pages 256–270, New York, NY, USA, 2016. ACM. doi:10.1145/2837614.2837655.

98 Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Position Paper: Performance Evaluation for Gradual Typing. In *Workshop on Script-to-Program Evolution (STOP)*, New York, NY, USA, 2015. ACM. URL: `http://www.ccs.neu.edu/home/types/publications/pe4gt/pe4gt.pdf`.

99 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 793–810, 2012. `doi:10.1145/2384616.2384674`.

100 Satish R. Thatte. Quasi-Static Typing. In *Principles of Programming Languages (POPL)*, pages 367–381, 1990. `doi:10.1145/96709.96747`.

101 Sam Tobin-Hochstadt, September 2018. URL: `https://twitter.com/samth/status/1039707471290478595`.

102 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 964–974, 2006. `doi:10.1145/1176617.1176755`.

103 Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Principles of Programming Languages (POPL)*, pages 395–406, New York, NY, USA, 2008. ACM. `doi:10.1145/1328438.1328486`.

104 Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *International Conference on Functional Programming (ICFP)*, pages 117–128. ACM, 2010.

105 Sam Tobin-Hochstadt, Matthias Felleisen, Robert Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory Typing: Ten Years Later. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *Summit on Advances in Programming Languages (SNAPL)*, volume 71, pages 17:1–17:17, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.SNAPL.2017.17`.

106 Sam Tobin-Hochstadt and Robert Bruce Findler. Cycles Without Pollution: A Gradual Typing Poem. In *Workshop on Script to Program Evolution (STOP)*, pages 47–57, New York, NY, USA, 2009. ACM. `doi:10.1145/1570506.1570512`.

107 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`.

108 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`.

109 Jesse A. Tov and Riccardo Pucella. Practical Affine Types. In *Principles of Programming Languages (POPL)*, 2011. `doi:10.1145/1926385.1926436`.

110 Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan. From fine- to coarse-grained dynamic information flow control and back. *PACMPL*, 3(POPL):76:1–76:31, 2019.

111 Niki Vazou. *Liquid Haskell: Haskell as a Theorem Prover*. PhD thesis, University of California, San Diego, 2016. URL: `https://escholarship.org/uc/item/8dm057ws`.

112 Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and Evaluation of Gradual Typing for Python. In *Symposium on Dynamic Languages (DLS)*, pages 45–56, New York, NY, USA, 2014. ACM. `doi:10.1145/2661088.2661101`.

113 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 *Principles of Programming Languages (POPL)*, pages 762–774, New York, NY, USA, 2017. ACM. `doi:10.1145/3009837.3009849`.

114 Philip Wadler and Robert Bruce Findler. Well-Typed Programs Can't Be Blamed. In *European Symposium on Programming (ESOP)*, pages 1–16, 2009. `doi:10.1007/978-3-642-00590-9_1`.

115 Stephanie Weirich. The influence of dependent types (keynote). In *Principles of Programming Languages (POPL)*, page 1. ACM, 2017.

116 Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Aaron Stump, Harley Eades, Peng (Frank) Fu, Garrin Kimmell, Tim Sheard, Ki Yung Ahn, and Nathan Collins. The Preliminary Design of the Trellys Core Language, 2011. Discussion session at PLPV.

117 Sam Tobin-Hochstadt with Jeremy Siek, Asumu Takikawa, Ben Greenman, et al. URL: `https://github.com/samth/gradual-typing-bib`.

118 Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual Typestate. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 459–483, 2011. `doi:10.1007/978-3-642-22655-7_22`.

119 Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *Principles of Programming Languages (POPL)*, pages 377–388. ACM, 2010.

120 Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. Consistent Subtyping for All. In Amal Ahmed, editor, *European Symposium on Programming (ESOP)*, pages 3–30, Cham, 2018. Springer International Publishing.

121 Zhe Yang. Encoding Types in ML-like Languages. In *International Conference on Functional Programming (ICFP)*, pages 289–300, New York, NY, USA, 1998. ACM. `doi:10.1145/289423.289458`.