Gradual Typing as if Types Mattered

RONALD GARCIA, University of British Columbia, Canada
ÉRIC TANTER, University of Chile, Chile and Inria Paris, France

Gradual typing is a principled approach to integrating static and dynamic type checking. To clarify what gradual typing means, Siek and collaborators articulated criteria for gradually typed languages: essentially that type checking and execution must smoothly transition between the two type checking regimes. However, these criteria say nothing about the relationship between the semantics of gradual and static types. This absence, which can be ascribed to gradual typing's tendency to focus on simple types, becomes critical for advanced type disciplines, where type *safety* does not imply type *soundness*. We assert a semantic criterion for gradual types. To coherently blend static and dynamic checking of a particular type discipline, a gradual type system must enforce the same invariants as those *intended* by its purely-static counterpart. We justify this stance by considering several type disciplines, highlighting key challenges and opportunities.

ACM Reference Format:
Ronald Garcia and Éric Tanter. 2020. Gradual Typing as if Types Mattered. In *Informal Proceedings of the first ACM SIGPLAN Workshop on Gradual Typing (WGT20).* ACM, New York, NY, USA, 11 pages.

1 INTRODUCTION

After decades of acrimony between proponents of statically typed languages and dynamic languages, an air of cooperation has swept the community. Programmers and language designers alike acknowledge that static and dynamic languages have complementary strengths, and many modern languages aspire to combine the perceived benefits of both styles while evading their shortcomings. This aspiration is not new, but a spate of recent industrial-strength languages has precipitated new approaches to designing hybrid languages and inspired researchers to enrich programming language theory to accommodate them [Savage 2014].

Arguably the most prominent current theory for designing such languages is gradual typing [Siek and Taha 2006]. It introduces new notions to programming language design, like *gradual types* and *consistency*, while building on prior techniques like *casts* and *coercions*. In addition to its technical aspects, gradual typing researchers have introduced correctness criteria and formal foundations, which together partially circumscribe what counts as a proper gradual language and how to extend its fundamental notions to new languages and type disciplines [Garcia et al. 2016; Siek et al. 2015].

Since gradual typing builds directly on static typing, it is only natural that it inherits decades of research into the design, evaluation, and implementation of statically typed languages. However, because of its integration of notions from dynamic languages, which were not developed with types in mind, gradual typing immediately highlights significant issues with our present understanding of type systems, issues that do not necessarily arise when designing purely static languages. In fact, these issues go to the very heart of type systems research, let alone gradual type systems, and echo earlier distinctions about type systems that have fallen from prominence but find new life in gradual typing.

Designing and characterizing principled gradual type systems forced us to revisit our understanding of *type safety* and *type soundness*, which are often conflated in the literature. This conflation causes no obvious technical problem for many type disciplines, but it becomes particularly critical as we consider what guarantees a gradual type system should provide to programmers, how those guarantees relate to those enforced by static and dynamic languages, and how these notions scale

WGT20, January 25,2020, New Orleans
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.

when gradualizing more sophisticated type disciplines. In fact, this distinction even expands the design space of type systems that have already been gradualized, as we demonstrate by example.

Following a brief introduction to the principles of gradual typing that have developed over the past decade, we highlight how certain fundamental concerns repeatedly arise when designing gradual type systems. We point to type *soundness* as a critical and often misunderstood notion that can be rehabilitated to address those concerns. To unpack these notions, we discuss several case studies that highlight how type soundness should be a critical guiding principle for gradually-typed languages.

2 GRADUAL TYPING IN A NUTSHELL

We briefly introduce gradual typing with the Gradually-Typed Lambda Calculus (GTLC) [Siek and Taha 2006]; though simple, it exhibits most of the fundamental ideas. GTLC is a pure extension of the Simply-Typed Lambda Calculus (STLC): every STLC program is a GTLC program, *e.g.*, $(\lambda f : \mathsf{Int} \to \mathsf{Int}.\ f\ 7)\ (\lambda x : \mathsf{Int}.\ x)$. GTLC's only syntactic addition is a new atomic type ?, called *the unknown type*, which represents the omission, or intentional obscuring, of static type information. It combines with static type information to form *gradual types*, like $? \to \mathsf{Int}$ and $\mathsf{Bool} \to ?$, which represent *imprecise* static type information. Thanks to ?, programs that cannot be typed in STLC can be typed and executed in GTLC, *e.g.*, $(\lambda x : ? \to ?.\ (x\ x)\ 7)\ (\lambda x : ?.\ x)$ evaluates to 7.

GTLC statically rejects programs with *local* type inconsistencies, *e.g.*, $(\lambda x : ? \to ?.\ x\ 7)\ 8$ is rejected since the number 8 could *never* have function type $? \to ?$. But type inconsistencies are only checked locally, so ? can be used to obscure type inconsistencies, *e.g.*, $(\lambda x : ? \to ?.\ x\ 7)\ (8 :: ?)$ ascribes the unknown type to 8, and the expression $8 :: ?$ could have *any* type so *might* have a function type. GTLC accepts this program, but signals a *runtime* type error at this function call. This ability to obscure type information enables GTLC to represent *any* program in the fully dynamic lambda calculus (DLC), which has no type annotations and statically rejects only syntactically ill-formed programs. For example, the (absurd) DLC program 7 3 is represented as the (still absurd) GTLC program $(7 :: ?)\ (3 :: ?)$.

Combining the benefits of static and dynamic languages was the initial inspiration for gradual typing, and the main force driving its development. But the concepts scale beyond *e.g.*, Python vs. OCaml. In fact, "static" and "dynamic" can be viewed as relative concepts, where *e.g.*, Scheme-style unityping is more dynamic than OCaml-style static typing is more dynamic than Liquid Haskell-style refinement typing [Vazou et al. 2014]. We now describe the principles of gradual typing, which undergird a variety of gradual type systems [Bañados Schwerter et al. 2016; Jafery and Dunfield 2017; Lehmann and Tanter 2017].

Formalities. Gradual types are partially ordered by a *precision* relation $\sqsubseteq$ directed along increasing imprecision, with fully static types as minimal elements. In GTLC, for instance, the chain $\mathsf{Bool} \to \mathsf{Int} \sqsubseteq \mathsf{Bool} \to ? \sqsubseteq ? \to ? \sqsubseteq ?$ spans from a fully-precise static type to the fully imprecise unknown type. Unlike subtyping, precision is covariant for function types because a ? in any position of a type indicates missing static type information.

GTLC's static semantics replaces type equality $T = T$ with the novel *consistency* relation $T \sim T$. Consistency generalizes type equality to gradual types, interpreting imprecise type information optimistically. It can be characterized in terms of precision: $T_1 \sim T_2$ holds for two *gradual* types if and only if $T_1 \sqsupseteq T_1' = T_2' \sqsubseteq T_2$ holds for some *static* types T_1', T_2'. In other words, consistency *lifts* static type equality along the precision relation. This lifting applies to other type relations, *e.g.* lifting subtyping to *consistent subtyping* [Garcia et al. 2016]. Since static types are minimal with respect to precision, it immediately follows that consistency (resp. consistent subtyping)

conservatively extends type equality (resp. subtyping): both relations are identical on static types. The conservative extension of static typing concepts is a recurring theme in gradual typing.

Since gradual programs omit some static type information, their dynamic semantics must account for gaps using runtime checks. Thus, a gradual program that successfully type checks may signal a runtime check error, just like a dynamic language program.

Criteria. Over time, researchers have established formal criteria that gradually-typed languages must satisfy [Siek and Taha 2006; Siek et al. 2015]. We briefly reprise those criteria, which we call *the STVCB criteria* after their progenitors.

First, a gradual language *conservatively extends* some pre-conceived "fully static" language. In GTLC's case that language is STLC: GTLC accepts and rejects exactly the same precisely-typed programs as STLC, and well-typed programs behave identically in both languages. Second, a gradual language *macro-expresses* some pre-conceived "fully dynamic" language: source programs can be desugared into the gradual language using only local compositional rewrites [Felleisen 1991]. For GTLC, that language is the dynamic calculus DLC. These criteria ensure that both extremes, fully-static and fully-dynamic, can be combined seamlessly in a single language. One can even define a *hybrid* language that blends STLC and DLC by straightforward translation to GTLC, which acts as a unifying runtime [Garcia and Cimini 2015].

The third STVCB criterion constrains how typability and execution vary as the precision of a program's type information varies. These *gradual guarantees* are formalized in terms of successful typing and execution, but are best understood in contrapositive form. The static gradual guarantee ensures that an *ill-typed* gradual program will remain *ill-typed* no matter which way its types are made *more* precise. The corresponding dynamic gradual guarantee ensures that a well-typed program that signals a runtime check error will continue to do so, or become ill-typed, no matter which way its types are made more precise. Both guarantees ensure that static and dynamic type errors indicate some fundamental inconsistency with the underlying static type discipline.

The STVCB criteria still leave open a substantial design space, even after the dynamic and static type disciplines are chosen [Siek et al. 2009]. This paper argues that this design space requires further *type-driven constraint* if gradual typing is to reach its full potential.

3 GRADUAL TYPING IS ABOUT TYPING

In the above discussion of gradual typing, we intentionally omit a stated criterion for gradually typed languages: that they must satisfy *type safety*, by which we mean that the semantics satisfy a Wright-Felleisen style Progress and Preservation theorem, that well-typed programs don't get stuck [Wright and Felleisen 1994]. This theorem is often called *type soundness*, as though the two notions are identical, and both are also associated with the tag line "well-typed programs don't go wrong [Milner 1978]." As observed by Siek et al. [Siek et al. 2015], this conception of "soundness" is unsatisfying for gradual typing because, though well-typed programs don't get stuck, they definitely produce runtime type errors, *i.e.*, they go wrong! In fact they must, otherwise GTLC could not embed all DLC programs. Indeed, progress and preservation generally guarantee only that the operational semantics is fully defined, *i.e.* that the only terminal runtime configurations are the intended ones. It is in this sense that a language like DLC can be deemed type safe, thanks to an especially boring type system [Harper 2012]!

Gradual typing researchers, ourselves included, have sought to clear this fly from the ointment by adapting notions of *blame* from the contracts community [Findler and Felleisen 2002]. We argue instead that much of this trouble can be addressed by recalling that type soundness and type safety are *not* identical notions but are often closely related: type safety often *implies* type soundness.

History. The first formalization of type soundness is due to Milner [Milner 1978], and though that paper does in fact prove that well typed programs don't go wrong, this fact is a *corollary* to the soundness theorem, not the theorem itself. Soundness for type systems draws directly from its analogue in mathematical logic, static analysis, and program logics. It states that, given a system of logical judgments, a syntactic proof system for those judgments, and an *intended* formal interpretation of those judgments, the proof system can prove a judgment only if its formal interpretation holds semantically.

In mathematical logic, the quintessential example, a judgment is typically interpreted in set-theoretic structures. Milner's insight was to interpret typing judgments using denotational semantics.[1] In modern notation, he defines a judgment $\Gamma \models t : T$ which is defined directly in terms of the dynamic semantics of programs. This *semantic* judgment is taken as the intended behavioral meaning of types. Then, he separately defines a *syntactic* judgment $\Gamma \vdash t : T$ essentially in the standard inductive way. Armed with these, the statement of semantic type soundness is simply "if $\Gamma \vdash t : T$ then $\Gamma \models t : T$", *i.e.*, the type checker guarantees interesting *local* behavioral properties of program fragments.

Milner's theorem guarantees that programmers can use types to reason about their programs in rich local ways, but nowadays, language researchers often recall the pithy corollary more than the theorem.[2] While the ability to eradicate an entire class of program bugs has great value, that property has not carried over to gradual typing intact. We believe that type soundness can, and should.

Is GTLC type sound? (take 1) On its own, this question does not make any sense, because as recalled above from Milner, a type system is sound *with respect to a given semantic judgment*. So before answering the question of whether GTLC (or any typed language for that matter) is sound, one ought to be explicit about the *intended meaning* of types. As it turns out, several such meanings can be formulated for any given typing discipline, even for a discipline as straightforward as simple types, as we will see shortly.

Once the intended meaning of types is clearly stated, one can explicitly identify the *type-based reasoning principles* that the type discipline enables. Type-based reasoning is a key asset for software engineering because it is: *i) modular*: one can reason about a function or module just based on its interface, independently of its implementation; *ii) compositional*: one can deduce new facts about a program by combining knowledge about its parts, in a syntax-directed manner; *iii) static*: one deduces information about the runtime behavior of a program without having to actually run it.

Reasoning with simple types. The simple typing discipline of STLC denotes the kinds of values that can be obtained by reducing an expression. Crucially, value classification can be interpreted in two ways, corresponding to *partial* or *total* correctness properties.[3] The partial interpretation says that *if* the reduction of a well-typed term terminates to a value, *then* that value has the predicted type. Formally:[4]

$$\text{If } \vdash t : T \text{ and } t \longrightarrow^* v \text{ then } v : T$$

The total interpretation is stronger, saying that a well-typed term *necessarily* terminates, and does so to a value of the predicted type. Formally:

[1] Milner and Tofte later adapted this to operational semantics [Milner and Tofte 1991].
[2] The original conception persists *e.g.* in work on foundational proof carrying code [Ahmed 2004].
[3] This nomenclature comes from program logics, where the distinction is common.
[4] For simplicity, we inline implications of type soundness, phrasing principles in terms of syntactic typing.

$$\text{If} \vdash t : T \text{ then } t \longrightarrow^* v \text{ and } v : T$$

This corresponds exactly to strong normalization proofs for STLC, which is an explicitly desired property in some contexts [Pierce 2002].

Additionally, the simple typing discipline offers reasoning principles on values, *i.e.* canonical form lemmas, that are valid regardless of the chosen interpretation regarding partial or total correctness. Examples of such reasoning principles are:

$$\text{If} \vdash v : \mathsf{Int} \text{ then } v = n, \text{ for some integer } n$$
$$\text{If} \vdash v : T_1 \to T_2 \text{ then } v = \lambda x : T_1.t, \text{ such that } x : T_1 \vdash t : T_2$$

These reasoning principles are exploited to achieve *safety* without requiring runtime tag checks before computational steps.

The fact that the STLC is strongly normalizing can be seen as a "side bonus" of the typing discipline, if one is mostly interested in avoiding stuck terms without having to resort to dynamic checks; this is by far the most common view on simple types, as presented in the classic literature on type systems [Pierce 2002]. Alternatively, strong normalization can be considered *a determining feature* of the discipline—and now safety-without-checks comes as a side bonus. Really the difference is in the eyes of the beholder: both perspectives are legitimate, and the language designer must set forth his or her intended principle.

Is GTLC type sound? (take 2) If simple types are meant to ensure *partial* correctness of value classification, then GTLC is sound. The runtime semantics of GTLC does ensure that primitive operators do not require exhaustive runtime checks; instead checks are only executed when necessary to ensure that statically-made assumptions are not violated. Conversely, if simple types are meant to ensure *total* correctness of value classification, then GTLC is unsound because GTLC is not strongly normalizing. Indeed, one can define and run (indefinitely) the term omega $(\lambda x : ?.x\ x)\ (\lambda x : ?.x\ x)$ in GTLC. While this distinction may seem contrived for purposes of programming languages, it represents a simple example of a circumstance where adding gradual types can fundamentally change the properties that types guarantee. We analyze more advanced examples in the next sections.

We observe that despite the attention that gradual typing has drawn, no termination-preserving gradual language has been developed. Recent work in the contract literature demonstrates that reasoning about termination can be codified and enforced using runtime monitors [Nguyễn et al. 2019]. As has often been the case with ideas from contracts, this technique may transfer smoothly to gradual typing. well for gradual typing Garcia *et al.* [Garcia et al. 2016] do mention the possibility of designing a language in which the unknown type only possibly denotes base types, but neither develop such a design nor relate it to the termination property.[5]

Type safety vs. type soundness. In any typing discipline where types specifically denote the class of values that a computation may return to the current program point—if the computation yields any such value at all—a type safety proof plus the canonical forms lemma implies soundness with respect to that interpretation. This logical implication explains the tendency for blurring the difference between type safety and type soundness. In fact, Wright and Felleisen [1994] observe a distinction between *weak* and *strong* soundness theorems, where the weak theorem directly corresponds to Milner's corollary and "establishes that a static type system achieves its primary goal of preventing type errors...". The paper takes this goal as the blueprint for syntactic type safety, but they note that "it is possible to demonstrate a stronger property that relates the answer produced to the type of the program." This stronger theorem corresponds to Milner's soundness

[5] Eremondi et al. [2019] use approximate normalization to ensure termination of compile-time evaluation at the type level.

theorem, though the paper deems classification a side benefit compared to proving that well-typed programs don't go wrong.

In addition to simple types, other examples of such "partial value-based classification" typing disciplines include subtyping, be it with records, objects, or even more semantic subtyping such as types refined with logical formulas [Rondon et al. 2008]. For many expressive typing disciplines, however, type safety *does not* imply type soundness, even when one is only concerned with partial correctness. The next two sections present concrete examples of typing disciplines where this disconnect arises: universally quantified polymorphic types and information-flow security types.

4 THE MEANING OF UNIVERSALLY QUANTIFIED TYPES

> *"I am convinced that a satisfactory model [of parametric polymorphism] should exclude ad hoc polymorphism."*
> – John Reynolds, 1983.

Universal quantification allows the definition of programs that are parametrized by types, using type abstraction ($\Lambda T.t$) and type application ($t\ [T]$). In addition to the partial/total correctness distinction, universal quantification admits two possible semantic interpretations.

Parametricity vs. genericity. Parametricity is the interpretation of type parameters that dictates that a polymorphic value must behave uniformly for all possible instantiations [Reynolds 1983]. This implies that one can derive theorems about the behavior of a program by just looking at its type, hence the name "free theorems" coined by Wadler [Wadler 1989]. For instance:[6]

$$\text{If} \vdash v : \forall\alpha.\alpha \to \alpha, \text{ then for all} \vdash v' : T, \text{ we have } v\ [T]\ v' \approx v'.$$

Languages with quantification that also support intensional type analysis or reflection, like Java, make it possible to define non-parametric functions. For instance, in Java, a generic method of type $\forall\alpha.\alpha \to \alpha$ can use `instanceof` to discriminate when α is in fact `String` and behave differently in such a case. For these languages, one can formulate a weaker property of quantified types that does not dictate uniform behavior. We call this property *genericity*, by analogy to the name generics in use in object-oriented languages like Java and C#.[7] Genericity simply captures the fact that a value of a quantified type can be instantiated at any type, and satisfy the resulting signature as expected. For instance:

$$\text{If} \vdash v : \forall\alpha.\alpha \to \alpha, \text{ then for all} \vdash v' : T, \text{ we have } v\ [T]\ v' : T.$$

Concretely, this means that if a value f has type $\forall\alpha.\alpha \to \alpha$, then genericity only tells us that f `[Int] 5` (if it terminates) reduces to some integer n, while parametricity tells us that f `[Int] 5` (if it terminates) reduces to 5.

Gradual genericity vs. gradual parametricity. Consider the following well-typed gradual program, in which the `add1` function is first given the unknown type, and then passed as argument to functions `g` and `h`, both of which expect a function of type $\forall\alpha.\alpha \to \alpha$:

```
let add1 : ? = fun x => x+1
let g : (forall a.a->a) -> Int = fun f => if (f [Bool] true) then 1 else 0
let h : (forall a.a->a) -> Int = fun f => f [Int] 0
(g add1) + (h add1)
```

[6] Contextual equivalence $\approx$ specifies that two expressions cannot be distinguished in any valid context.
[7] The literature typically uses genericity and parametricity interchangeably, while for Java or C++ programmers, genericity does not suggest the strong meaning of parametricity.

The g function instantiates its argument at type Bool, while h instantiates it at type Int. When executing g add1, a runtime cast error should be reported, in order to ensure *safety*. This error reflects the fact that add1 is *not* a generic function.

In contrast, h add1 is safe, because add1 is an Int → Int function. Whether or not this application violates soundness depends on the intended meaning of quantified types. If one is interested only in genericity, then h add1 should execute without errors. If one expects parametricity, though, a runtime violation should be reported: the type-based reasoning principle induced by parametricity means that the programmer can legitimately expect any function of type $\forall\alpha.\alpha \to \alpha$ (which is fully precise) to be equivalent to the identity function.

Ahmed *et al.* first identified the challenges of respecting parametricity when introducing the unknown type [Ahmed et al. 2011]. They introduce a cast calculus, not a gradual source language, and have not formally established parametricity (it remains a conjecture). Cimini and Siek [Cimini and Siek 2017] automatically derive a gradual language with universally quantified types, but only prove type safety, so the gradual language satisfies (what we call) genericity, not parametricity. Igarashi et al. [2017] introduce a polymorphic gradual calculus which, by safety, satisfies genericity. Parametricity and the dynamic gradual guarantee are left as future work. More recently, Ahmed et al. [2017] defines a polymorphic blame calculus that satisfies a relational parametricity theorem, and demonstrate by example the utility for parametricity and their approach to achieving it. Building on this work, Toro et al. [2019] define a gradually typed variant of System F which satisfies parametricity, but in doing so highlights a possibly fundamental tension between parametricity and the dynamic gradual guarantee. While progress and preservation are proven in the latter two works, these theorems are not the basis for proving soundness. Each uses *step-indexed logical relations*, an extension of techniques used in Reynolds [1983] to prove parametricity. These models associate types with *pairs* of program fragments, so as to formalize rich behavioral invariants like parametricity. Semantic typing for programs is defined in terms of this binary relation, and syntactic typing is proven sound with respect to semantic typing. Ahmed [2004] provides a clear and motivating introduction to this technique.

5 THE MEANING OF SECURITY TYPES

Information-flow security typing enables statically classifying program entities with respect to their confidentiality levels, expressed via a lattice of security labels [Denning 1976]. For instance, a two-level lattice $\mathsf{L} < \mathsf{H}$ allows distinguishing public or low data (*e.g.* Int_L) from confidential or high data (*e.g.* Int_H). An information-flow security type system statically ensures *noninterference*: high-confidentiality data may not flow directly or indirectly to lower-confidentiality channels [Volpano et al. 1996]. To do so, the security type system tracks the confidentiality level of computation based on the confidentiality of the data involved. Note that similar to total vs. partial correctness, a security type system can be designed to guarantee either termination-*sensitive* noninterference or, more commonly, termination-*insensitive* noninterference.

Like parametricity, the reasoning principles supported by security typing dictate precise knowledge about the (noninterfering) behavior of functions. For instance:

$$\text{If } \vdash v : \mathsf{Int}_\mathsf{H} \to \mathsf{Int}_\mathsf{L}, \text{ then for all } \vdash v_1, v_2 : \mathsf{Int}_\mathsf{H}, \text{ we have } v\ v_1 = v\ v_2 \text{ (if both terminate).}$$

In words, because the argument is a secret and the return value is public, the result of applying the function cannot possibly depend on its argument.

Disney and Flanagan [Disney and Flanagan 2011], and then Fennell and Thiemann [Fennell and Thiemann 2013], designed security-typed languages dubbed gradual. They are both, however, not gradual in the sense of the STVCB criteria (Sect. 2), because they are not based on precision and the corresponding notion of consistency. In these languages, security types do not denote

noninterference guarantees; they denote *upper bounds* on value labels. For instance, given a function of type $\mathsf{Int}_\mathsf{H} \to \mathsf{Int}_\mathsf{L}$, the only guarantee is that the argument is *at most* private; *i.e.* it can also be public. Surprisingly, Int_L as a type is more informative than Int_H. Consequently, the function can very easily reveal its argument, using a runtime security cast: $\lambda x.\langle \mathsf{Int}_\mathsf{L} \rangle x$. At runtime, if the argument value is labeled H, a runtime error is raised; if the argument is labeled L, the function behaves as the identity, thereby violating the noninterference reasoning principle stated above. This situation parallels the previous discussion about gradual parametricity. In these languages, "value-label-based" noninterference is guaranteed by the runtime system; type-based reasoning provides *no* static information about noninterference. Toro et al. [2018] introduces a gradually-typed security language where gradual security types are fundamentally connected to modular noninterference guarantees as well as to Siek et al. [2015]-style precision. Once again, progress and preservation are proven for the type system in question, but the semantic soundness of security types is established using step-indexed logical relations. The language achieves noninterference at the expense of the dynamic gradual guarantee, and it remains an open question whether and to what extent a language with the present feature set can achieve both simultaneously.

6 DESIGNING SOUND GRADUAL LANGUAGES

Designing a sound gradual language is not free of challenges, especially when the intended meaning of types is a rich semantic property.

Recently, a couple of approaches have addressed the question of systematizing the design of gradual languages. For instance, the Gradualizer aims at the automatic construction of a gradually-typed language from the definition of statically-typed one [Cimini and Siek 2016, 2017]. The Gradualizer's scope, however, is restricted to adding *only* the unknown type to value-based type disciplines.

Another approach, wider in scope but not automatic, is Abstracting Gradual Typing (AGT) [Garcia et al. 2016]. AGT exploits an understanding of gradual types through the eyes of abstract interpretation [Cousot and Cousot 1977] to allow for the systematic derivation of gradual languages. AGT has been applied to typing disciplines outside the scope of the Gradualizer, such as effects [Bañados et al. 2014; Bañados Schwerter et al. 2016], refinement types [Lehmann and Tanter 2017], and the type disciplines of the previous two sections.

Both the Gradualizer and AGT do not directly address the soundness of the gradual language with respect to the intended meaning of types. For instance, applying either approach to STLC yields GTLC, which is non-terminating. Because soundness of refinement types is value-based, applying AGT to a language with refinements yields a sound gradual language [Lehmann and Tanter 2017]. However, in the case of security typing, we have shown that noninterference is lost when the language includes references. A similar situation seems to occur with parametric polymorphism: applying either methodology preserves genericity, but not the type-based reasoning expected from parametricity. Recovering type-based reasoning demands ingenuity.

Recent work demonstrates that even gradual languages with value-classifying types benefit from formalizing the semantics of types as (binary) relational properties. New and Ahmed [2018] and New et al. [2019] construct semantic models of simple gradual types to more easily prove the gradual guarantees for bespoke language designs and to justify type-based program optimizations. This line of work points toward further practical and conceptual benefits for focusing on the semantics of gradual types.

The current state-of-the-art in gradual language design is therefore greatly limited with respect to type soundness; we believe that this is a major open challenge for gradual typing research. This soundness challenge interferes with the expected graduality of the language. For instance, if one would be willing to restrict gradual types to denote only base types, the resulting gradual language

would trivially be terminating; however, such a gradual language could not macro-express DLC anymore (Sect. 2).

Finally, other recent work points to shortcomings in the present criteria for gradual languages and propose new criteria that we believe complement the consideration of type soundness. Greenman et al. [2019] adapt the concept of *complete monitoring* from the behavioral contract literature to gradual typing [Dimoulas et al. 2012]. They find that doing so provides another means for drawing meaningful distinctions between the semantics of different gradual counterparts to the same static type discipline.

7 CONCLUSION

We believe that it is important to develop a firm understanding of the pragmatic and theoretical issues facing gradual typing if its ideas are to successfully make their way into mainstream languages. In particular, a proper generalization of gradual typing concepts can make complex type disciplines usable in practice, by lowering the barrier to entry. However, for this to happen, we believe it's important to have a strong understanding of the conceptual challenges that gradual typing poses to our current understanding of how to design and evaluate typed programming languages.

This submission distills some of our recent experience with designing and evaluating type systems, focusing on how researchers think about types and what they bring to the table. Our recent struggles with designing gradual type systems for increasingly sophisticated type disciplines has uncovered some notable challenges in our understanding that must be addressed in order to move forward. We find that certain simplifications (notably the conflation of type safety and type soundness), which have been very fruitful for a long time, now haunt us when we try to understand whether a gradual type system "makes sense." Siek et al. [2015] already aimed at more precisely characterizing what makes a language gradual. We believe it was a valuable first step, and facilitated a substantial stream of advances in the gradual typing literature, as well as our understanding of how to analyze and evaluate gradually typed language designs. We believe that substantial conceptual gains are to be had by moving beyond type safety, which insufficiently foregrounds the broader importance of considering the intended semantics of types.

Designing a gradually-typed language means much more than throwing in dynamic checks and making sure that programs do not get stuck. We believe gradual language designers should be very explicit about their intent with respect to both the spectrum of graduality—*i.e.* clearly specifying what type disciplines bracket both ends—as well as with respect to the meaning of types—*i.e.* what reasoning principles are preserved in the gradual language.

However, because most type systems focus on partial correctness for value classification, the gradual typing community has tended to take type-based reasoning principles for granted, leaving them un(der)specified. We have shown through three different examples that there are significant semantic issues involved, which challenge current language designs.

In fact, our analysis calls for a type-soundness analogue to the gradual guarantee, capturing the continuity of type-based reasoning with type precision. For instance, in security typing, when given a function of type $\mathsf{Int}_{\mathsf{H}} \to \mathsf{Int}_{?} \to \mathsf{Int}_{\mathsf{L}}$, one cannot say whether the second argument influences the result, but one must be firm about the first argument.

Finally, while there has been steady progress in systematizing the design of safe gradual languages that satisfy the criteria of Siek *et al.*, much remains to be done to understand how to design gradual languages that are sound with respect to rich, possibly not value-based, semantics of types—if at all possible. For instance, the AGT methodology is built on type safety proofs; gradual languages derived through AGT inherit safety from this approach. Extending AGT to address soundness might require adapting the whole conceptual framework to take the purely static type soundness proofs as a source of design insight.

REFERENCES

Amal Ahmed. 2004. *Semantics of Types for Mutable State.* Ph.D. Dissertation. Princeton University.

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for All. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*. ACM Press, Austin, Texas, USA, 201–214.

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free: Parametricity, with and Without Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 39 (Aug. 2017), 28 pages.

Felipe Bañados, Ronald Garcia, and Éric Tanter. 2014. A Theory of Gradual Effect Systems. In *Proceedings of the 19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*. ACM Press, Gothenburg, Sweden, 283–295.

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2016. Gradual Type-and-Effect Systems. *Journal of Functional Programming* 26 (Sept. 2016), 19:1–19:69.

Matteo Cimini and Jeremy Siek. 2016. The gradualizer: a methodology and algorithm for generating gradual type systems, See [POPL 2016 2016], 443–455.

Matteo Cimini and Jeremy Siek. 2017. Automatically Generating the Dynamic Semantics of Gradually Typed Languages, See [POPL 2017 2017]. To appear.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL 77)*. ACM Press, Los Angeles, CA, USA, 238–252.

Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (May 1976), 236–243.

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings.* 214–233.

Tim Disney and Cormac Flanagan. 2011. Gradual information flow typing. In *International Workshop on Scripts to Programs.*

Joseph Eremondi, Éric Tanter, and Ronald Garcia. 2019. Approximate Normalization for Gradual Dependent Types. *Proceedings of the ACM on Programming Languages* 3, ICFP (Aug. 2019), 88:1–88:30.

Matthias Felleisen. 1991. On the expressive power of programming languages. *Science of Computer Programming* 17, 1 (1991), 35 – 75.

Luminous Fennell and Peter Thiemann. 2013. Gradual Security Typing with References. In *Proceedings of the 26th Computer Security Foundations Symposium (CSF)*. 224–239.

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *Proceedings of the 7th ACM SIGPLAN Conference on Functional Programming (ICFP 2002)*. ACM Press, Pittsburgh, PA, USA, 48–59.

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*. ACM Press, Mumbai, India, 303–315.

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing, See [POPL 2016 2016], 429–442.

Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2019. Complete monitors for gradual types. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 122:1–122:29.

Robert Harper. 2012. *Practical Foundations for Programming Languages.* Cambridge University Press.

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On Polymorphic Gradual Typing. *Proc. ACM Program. Lang.* 1, ICFP, Article 40 (Aug. 2017), 29 pages.

Khurram A. Jafery and Joshua Dunfield. 2017. Sums of Uncertainty: Refinements Go Gradual. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 804–817.

Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types, See [POPL 2017 2017], 775–788. To appear.

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17 (Aug. 1978), 348–375.

Robin Milner and Mads Tofte. 1991. Co-induction in Relational Semantics. *Theor. Comput. Sci.* 87, 1 (Oct. 1991), 209–220.

Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs. *Proc. ACM Program. Lang.* 2, ICFP, Article Article 73 (July 2018), 30 pages.

Max S. New, Daniel R. Licata, and Amal Ahmed. 2019. Gradual Type Theory. *Proc. ACM Program. Lang.* 3, POPL, Article Article 15 (Jan. 2019), 31 pages.

Phúc C. Nguyễn, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2019. Size-Change Termination as a Contract: Dynamically and Statically Enforcing Termination for Higher-Order Programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 845–859.

Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press, Cambridge, MA, USA.

POPL 2016 2016. *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. ACM Press, St Petersburg, FL, USA.

POPL 2017 2017. *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. ACM Press, Paris, France. To appear.

John C. Reynolds. 1983. Types, abstraction, and parametric polymorphism. In *Information Processing 83*, R. E. A. Mason (Ed.). Elsevier, 513–523.

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM Press, 159–169.

Neil Savage. 2014. Gradual Evolution. *Commun. ACM* 57, 10 (Sept. 2014), 16–18.

Jeremy Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the Design Space of Higher-Order Casts. In *Proceedings of the 18th European Symposium on Programming Languages and Systems (ESOP 2009) (Lecture Notes in Computer Science)*, Giuseppe Castagna (Ed.), Vol. 5502. Springer-Verlag, York, UK, 17–31.

Jeremy Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Proceedings of the Scheme and Functional Programming Workshop*. 81–92.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. 274–293.

Matías Toro, Ronald Garcia, and Éric Tanter. 2018. Type-Driven Gradual Security with References. *ACM Trans. Program. Lang. Syst.* 40, 4, Article 16 (Dec. 2018), 55 pages.

Matías Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual Parametricity, Revisited. *Proc. ACM Program. Lang.* 3, POPL, Article 17 (Jan. 2019), 30 pages.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 269–282.

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4, 2-3 (Jan. 1996), 167–187.

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*. ACM, New York, NY, USA, 347–359. https://doi.org/10.1145/99370.99404

Andrew K. Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Journal of Information and Computation* 115, 1 (Nov. 1994), 38–94.