Gradually Typed Languages Should Be Vigilant!

OLEK GIERCZAK, Northeastern University, USA
LUCY MENON, Northeastern University, USA
CHRISTOS DIMOULAS, Northwestern University, USA
AMAL AHMED, Northeastern University, USA

In gradual typing, different languages perform different dynamic type checks for the same program even though the languages have the same static type system. This raises the question of whether, given a gradually typed language, the combination of the translation that injects checks in well-typed terms and the dynamic semantics that determines their behavior sufficiently enforce the static type system of the language. Neither type soundness, nor complete monitoring, nor any other meta-theoretic property of gradually typed languages to date provides a satisfying answer.

In response, we present *vigilance*, a semantic analytical instrument that defines when the check-injecting translation and dynamic semantics of a gradually typed language are adequate for its static type system. Technically, vigilance asks if a given translation-and-semantics combination enforces the complete *run-time typing history* of a value, which consists of all of the types associated with the value. We show that the standard combination for so-called Natural gradual typing is *vigilant* for the standard simple type system, but the standard combination for Transient gradual typing is not. At the same time, the standard combination for Transient is *vigilant* for a tag type system but the standard combination for Natural is not. Hence, we clarify the comparative type-level reasoning power between the two most studied approaches to sound gradual typing. Furthermore, as an exercise that demonstrates how vigilance can guide design, we introduce and examine a new theoretical static gradual type system, dubbed *truer*, that is stronger than tag typing and more faithfully reflects the type-level reasoning power that the dynamic semantics of Transient gradual typing can guarantee.

CCS Concepts: • **Software and its engineering** → **Semantics**.

Additional Key Words and Phrases: gradual typing, semantics, logical relations, natural, transient

ACM Reference Format:
Olek Gierczak, Lucy Menon, Christos Dimoulas, and Amal Ahmed. 2024. Gradually Typed Languages Should Be Vigilant!. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 125 (April 2024), 29 pages. https://doi.org/10.1145/3649842

1 VIGILANCE, A NEW ANALYTICAL INSTRUMENT FOR GRADUAL TYPING

Gradual typing exhibits an impressive variety. While industrial approaches tend to ignore types after static type checking, academic ones prioritize soundness and translate types into different forms of checks. Furthermore, how these checks behave varies greatly from design to design.[1] As a notable point of contrast, languages that adhere to Natural gradual typing [Matthews and Findler 2009; Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006] translate types to contract-enforcing proxies; while languages that adhere to Transient gradual typing [Vitousek et al. 2017] simply

[1]See Greenman et al. [2023] for a detailed overview of the landscape.

Authors' addresses: Olek Gierczak, Northeastern University, Boston, USA, gierczak.o@northeastern.edu; Lucy Menon, Northeastern University, Boston, USA, semiotics@ccs.neu.edu; Christos Dimoulas, Northwestern University, Evanston, USA, chrdimo@northwestern.edu; Amal Ahmed, Northeastern University, Boston, USA, amal@ccs.neu.edu.

ACM 2475-1421/2024/4-ART125
https://doi.org/10.1145/3649842

inject and perform so-called dynamic tag checks at strategic spots in a program. Recently, language designers have even been experimenting with ways to combine both approaches to gradual typing in a single language [Greenman 2022; Greenman et al. 2022]. This variety has a pragmatic background: sound gradual typing often incurs prohibitive performance costs [Greenman et al. 2019b], thus different design points derive from different ways of resolving the tradeoff between performance and guarantees. What this design space lacks, though, is a toolbox for analyzing what checks are necessary for type-based guarantees.

In response, this paper contributes *vigilance*, a new semantic analytical instrument for gradual typing. Vigilance captures an intuitive fact; when a language designer tweaks the translation from types to checks and/or the dynamics of a language, but leaves the static type system as is, parts of the types of a program may go unchecked, which can invalidate standard type-based reasoning principles.[2] If that is the case, there is an alternative type system that the modified language *actually enforces*, which can serve as the basis of type-based optimizations and refactorings. Vigilance reifies this adequate design point: a combination of translation and semantics is vigilant for a type system if the combination enforces all the types ascribed to the values produced during the evaluation of an expression.

Vigilance sends a pragmatic signal to language designers. When the translation and semantics of a language are not vigilant for its type system, the designer may want to consider a type system with less precise types, either as an alternative to the original one, or as the basis of semantics-preserving optimizations and IDE tools. Conversely, when the translation and semantics are vigilant for the language's type system, the language designer can use vigilance as a compass to find either a stronger type system, or a translation-and-semantics combination that results in fewer checks. En route to achieving these goals, vigilance offers language designers the instrument to determine whether the translation and dynamics are (in)adequate for the statics of a language.

Vigilance builds on prior research efforts to create such an instrument for gradual typing [Greenman et al. 2023; Greenman and Felleisen 2018; Greenman et al. 2019a; Siek et al. 2015a; Tobin-Hochstadt and Felleisen 2006; Vitousek et al. 2017; Wadler and Findler 2009] and improves on them. Specifically, vigilance subsumes two desired properties for gradual typing: type soundness and complete monitoring [Dimoulas et al. 2012; Greenman et al. 2019a]. Vigilance is stronger than both syntactic type soundness and its semantic counterpart: it asks not only that a program value behaves according to its *type*, but also to its *run-time typing history*. Consequently, vigilance enables developers to rely on types while they reason about dynamically typed code, which type soundness does not entail [Greenman et al. 2023, 2019a]. Vigilance is also both more fine-grained and stronger than complete monitoring: (i) it can positively or negatively characterize translation-semantics combinations other than the standard one of the Natural approach; and (ii) it entails that the gradually typed language performs the meaningful checks at the right times. §2 compares vigilance with type soundness and complete monitoring through concrete examples, and thus, demonstrates vigilance's value in a realistic setting. As we discuss in detail in §6, compared to other previously proposed desired properties for gradual typing, vigilance is not intended to replace them but to complement them.

Besides developing vigilance, in this paper we evaluate vigilance:

- Revisiting the literature, we perform a side-by-side analysis of the two most-studied sound approaches to gradual typing, Natural and Transient. Specifically, we focus on the canonical constituents of Natural and Transient gradual typing from the literature. For the Natural approach, these are: a *simple* type system, a *Higher-Order* (HO) translation that places proxy-generating casts in the appropriate places in a gradually typed program, and the so-called

[2]Hereafter, *type system* refers to a language's static type system, *translation* to the translation that turns types to checks, and *semantics* to the language's dynamic semantics.

Natural semantics that uses the proxies to enforce type-like properties at runtime. For the Transient approach, these are: a *simple* type system, a *First-Order* (FO) translation that places tag-enforcing casts and checks in the appropriate places in a gradually typed program, and the so-called Transient semantics that performs the corresponding tag enforcement at runtime. Inspired by prior syntactic analytical work [Greenman et al. 2023; Greenman and Felleisen 2018; Greenman et al. 2019a], in order to make an apples-to-apples comparison between the two seemingly radically different approaches, we devise a uniform framework that distills the differences between the two approaches to differences between their corresponding semantics. Hence, we show that to compare the two approaches, it suffices to compare the Natural and Transient semantics.[3]. Using our framework, we confirm prior results about the relative guarantees that the canonical formulations of the two approaches offer. That is, we prove that the combination of the HO translation and the Natural semantics is vigilant for the simple type system, while the combination of the FO translation and the Transient semantics is not.

- Going beyond prior work, we establish a surprising new fact about Natural and Transient. Namely, we show that even though the FO-Transient combination is vigilant for a *tag* type system, the HO-Natural one is not. Even though, from the perspective of the prior syntactic analysis [Greenman et al. 2023; Greenman and Felleisen 2018; Greenman et al. 2019a], Natural gradual typing is stronger than Transient gradual typing, from the semantic perspective of vigilance this is not the case; each corresponds to a different adequate design point. In particular, while the prior syntactic analysis seems to support the intuitive view that the Transient approach to gradual typing "forgets" some necessary checks, vigilance shows that the semantic reality is subtler than that. To enforce the types of function arguments, the Natural approach relies on checks that the HO translation places carefully at application sites that constitute the boundary between less and more precisely-typed expressions. In contrast, the Transient approach adopts an open-world stance and translated functions partially check their arguments themselves, which offers some protection in every context. Put differently, vigilance reveals an innate non-trivial distinction between the two approaches, which affects the assumptions programmers can make when programming with one or the other.
- Seeking to understand the reasoning power of the Transient approach, we use vigilance to perform a design exercise. Specifically, we construct an alternative translation (Flow) and a flow-sensitive type system, which we dub *truer*, that assigns to programs more precise types than the tag type system. We prove that the the combination of the Flow translation and the Transient semantics is vigilant for the truer type system, and hence, we establish that the tag checks that the Transient semantics perform are sufficient to deduce more type-level facts about program values than their type tag. Furthermore, to demonstrate the potential benefits revealed by the exercise, we use the truer type system to justify and prove correct an optimization that elides unnecessary dynamic type checks.

Outline. The remainder of the paper is organized as follows. §2 describes the necessary background, and the key ideas behind vigilance. §3 presents the formal linguistic framework of the paper and §4 builds on that to define vigilance formally and prove that the HO-Natural combination is vigilant for simple typing. §5 develops the tag and truer type systems, shows that FO-Transient is vigilant for tag typing and truer typing, and proves that truer renders unnecessary some of the checks that the Transient semantics performs. §6 describes related work not already covered in §2, and discusses future directions and some concluding thoughts.

[3]Because of our uniform framework, we can use Natural and Transient to refer to both the two semantics and the overall approaches, i.e., the type system together with a translation and a semantics. When the distinction matters, we use the words "approach" and "semantics" to clarify.

2 MOTIVATION AND THE MAIN IDEAS BY EXAMPLE

This section informally discusses the two approaches to gradual typing, Natural and Transient, that are the focus of this paper in a unified framework that makes an apples-to-apples comparison possible. The discussion, which revolves around a series of examples, also clarifies the shortcomings of prior work on type soundness and complete monitoring, and serves as the substrate for a high-level introduction of the technical ideas behind vigilance. In subsequent sections, we give these ideas a formal treatment.

2.1 Natural and Transient Gradual Typing in one Framework

At first look, Natural and Transient seem vastly different. Natural relies on proxies that are difficult [Greenman et al. 2019b] (though not impossible [Kuhlenschmidt et al. 2019]) to implement in a performant manner, while Transient offers a lightweight alternative where tag checks are in-lined at the start of function bodies, and around elimination forms. These differences raise the problem of a fair comparison between the two approaches. As we discuss briefly in §1, to overcome this obstacle, we follow Greenman and Felleisen [2018] and construct a linguistic setting that minimizes their differences to just the essentials. First, we decouple the syntax and static type system of the *source language*, which we refer to as the Gradually Typed Language (*GTL*) and which is based on the gradually typed λ-calculus [Siek and Taha 2006], from the meaning of its programs. GTL programs obtain meaning via *type-preserving* translations to an *intermediate language* that comes with both casts and type assertions, (*ICTL*).[4] Second, we define a unified translation, Uni, from GTL to ICTL that consolidates the differences between the HO and FO translations from the literature. In particular, Uni injects in the image of a GTL program all the casts that both Natural and Transient require, and the type assertions that are specific to Transient. Third, we construct a parameterized reduction semantics for ICTL that, for different parameters, matches either the Natural or the Transient semantics. Specifically, depending on the parameters, the casts and type assertions of an ICTL program either generate proxies, or simply perform tag checking, or act as trivial no-ops. As a result, this uniform setting allows us to study the differences between the Natural and Transient approaches described in the literature by focusing on the differences between the Natural and Transient variants of the ICTL semantics. In other words, for a given GTL program, only the evaluation of its ICTL image differs between the two approaches; everything else is the same.

Fig. 1 gives a taste of our GTL as the starting point of the discussion of the Natural approach. The snippet defines `segment_png`, which expects as its argument a PNG and then applies `segment`, which has no type annotations. Even though the static checker for the canonical simple gradual type system does not have at its disposal the necessary type information to derive that `segment` indeed has type $\mathsf{PNG} \to \mathsf{PNG} \times \mathsf{PNG}$, it accepts the program as well typed. To compensate for the partial static type checking, the HO translation of `segment_PNG` injects a cast around `segment` from the dynamic type $*$[5] to the expected type $\mathsf{PNG} \to \mathsf{PNG} \times \mathsf{PNG}$. Under the Natural semantics, the role of the cast is to check whether `segment` behaves as a function from PNGs to pairs of PNGs whenever it is called. Hence, the Natural cast checks that `segment` is a function and wraps it in a proxy that defers the remaining checks until `segment_png` is called.

```
let segment = λ img. (...)
let segment_png =
  λ (img:PNG) → PNG×PNG.
    segment img
```

Fig. 1. Dynamically Typed Argument for a Typed Parameter

[4] ICTL, our Intermediate Cast-and-Type-assertion Language, plays the role of cast calculi from the literature. Since the term "cast calculus" is overloaded and different variants have subtly different features, we introduce new nomenclature to avoid conflating terminology.

[5] Code without annotations implicitly has type $*$.

Library	Typed Wrapper	User Code

let segment = λ img. (...)
let crop = λ img. (...)
...
	let crop_png =
	  λ(img:PNG) → PNG. crop img
	let segment_png =
	  λ (img:PNG) → PNG×PNG.
	  segment img
	let segment_png_small =
	  λ (img:PNG) → PNG×PNG.
	  let (a, b) = segment img
	  (crop_png a, crop_png b)
		let write_img = λ img, path. (...)
		...
		let foreground =
		  (segment_png my_png)[0]
		let foreground_small =
		  (segment_png_small my_png)[0]
		write_img foreground "fg.png"
		write_img foreground_small "fgs.png"
		...

Fig. 2. Using a Typed Wrapper for a Dynamically Typed Image Library

The same GTL program behaves differently under the Transient approach. After type checking, similar to the HO translation, the FO one injects a cast around segment, however this cast is much weaker. The Transient cast only checks that segment is a function and does not create a proxy. To counter for the absence of the proxy, the FO translation further re-writes the code and injects type assertions to ensure that the results of any calls to segment_png are pairs.

While our exposition of the example above alludes to two different translations — one with higher-order casts that produce proxies (HO), and one with first-order casts and type assertions (FO) — as mentioned above, our framework employs a consolidated translation Uni from GTL to ICTL. The ICTL image of a GTL program has both casts and type assertions whose behavior depends on the parameterization of the ICTL reduction rules. As a result any ICTL program can run either in a Natural or a Transient manner: in the first case the casts behave as higher-order proxies and the assertions are no-ops; in the second, casts and assertions perform immediate first-order checks. Hence, the differences in the Natural and Transient behavior of the running GTL example boil down exactly to how the Natural and Transient semantics treat casts and assertions, which enables the apples-to-apples comparison of the two approaches. Interestingly, for the example in this section, even though the Transient semantics performs just lightweight tag checks and uses no proxies, it seems to entail the same type-level facts for the example as the Natural semantics with its costly proxies.

2.2 The Gap Between Statics and Dynamics for Transient

The actual difference in the type-level guarantees offered by Transient and Natural becomes clear when dynamically typed code uses a function with type other than *. Greenman et al. [2019a] demonstrate the issue with a scenario where a library with a typed interface is in fact dynamically typed and the interface is nothing more than a thin veneer of possibly misleading type annotations. Such scenarios are particularly important when languages first obtain gradual type systems; adding a type interface on top of a dynamically typed library offers to language designers a way to quickly grow the set of "typed" libraries, and hence, it encourages the further use of the gradual type system. But, when developers switch to these "typed" libraries in the process of type migrating their code, they may be surprised. Spefically, phrased in terms of our uniform framework, for such a scenario, the casts and assertions injected through the Uni translation of a GTL program to ICTL may not result in all the checks that are necessary to validate that the dynamically typed library adheres to its type interface; it all depends on the specific semantics of the ICTL. As a result, optimizations or refactorings that one might expect to hold if the types were enforced may not be valid.

To make the discussion concrete, fig. 2 conveys the scenario from Greenman et al. [2019a] in our GTL. In this example, dynamically typed code (right) uses segment_png and segment_png_small from a seemingly typed library (center) to split a PNG into an uncropped and cropped foreground and background respectively, and then writes each foreground to a file. However, the typed library

is just a thin typed wrapper around the dynamically typed implementation (left). In more detail, the actual dynamically typed implemantation provides two functions: `crop`, which crops images to a particular size, and `segment` which segments an image into a pair of a "foreground" and a "background" image. The typed library imports these two functions, and re-exports them: `crop_png` is a wrapper for `crop`, and `segment_png` and `segment_png_small` are wrappers for `segment`.

The scenario behaves differently under the Natural and Transient ICTL semantics. The Uni translation introduces casts at uses of `segment_png` and `segment_png_small` in the user code, but importantly, since the user code is dynamically typed, Uni does not inject any type assertions for the results of the projections in the body of `foreground` and `foreground_small`. Hence, the actual dynamic type checks during the evaluation of the example depend exactly on how the semantics treats the aforementioned injected casts. In the Natural semantics, the casts result in proxies that inspect the results of the function calls, ensuring the type from the annotation; in the Transient approach, the cast devolves to just a simple tag check of whether the result is a pair. As a result, in contrast to the Natural semantics the Transient semantics does not check whether the first component of the result of `(segment_png my_png)` is a PNG.

2.3 Type Soundness is Not Enough

At first glance, this difference between the Natural and the Transient semantics seems like an issue that type soundness should clarify. Type soundness does distinguish between the two [Greenman and Felleisen 2018], but falls short of fully characterizing the difference. Intuitively, in Transient, the use context of a function is expected to inspect the results of such a dynamically typed function via type assertions at call sites to make sure the function behaves as its type describes. Since, for the final result of a program there is no use context, syntactic type soundness for Transient can say little about programs that produce functions. In contrast, in Natural, the proxy around a function allows syntactic type soundness to establish that the function's results adhere to their type.

In formal terms, syntactic type soundness says that if a source program is well typed at type τ then when its ICTL image evaluates to a value with the Natural semantics, the value also has type τ. In contrast, when the ICTL image is run with the Transient semantics, the result value has a type that matches the tag of τ (i.e. its top type constructor), but that is not necessarily exactly equal to τ (tag soundness). Hence, syntactic type soundness seems to reveal that the choice of translation and ICTL semantics affects the predictive power of the GTL's simple type system.

However, syntactic type soundness stops short of explaining whether either the combination of the translation and the Natural or Transient ICTL semantics results in all the checks the GTL simple type system relies on. Syntactic type soundness only connects the type of the source program with the type of the translated ICTL program's result. Therefore, all intermediate types that do not contribute to that goal are immaterial. The running scenario demonstrates this situation. First, given the dynamically-typed nature of the user code, all occurrences of `segment_png` on the rightmost portion of the fig. are considered by the type system to be of type $*$, which is inhabited by all values, not just functions from PNGs to pairs of PNGs. So type soundness says nothing about these occurrences of `segment_png` and `segment_png_small` directly. Second, one cannot rely on the compositional nature of the typing rules of the type system and type soundness to deduce transitively some more precise type-level information for these occurrences than $*$. In fact, the rules for the simple type system, type soundness, and the details of the Uni translation and the Natural semantics all together are necessary to reason that `(segment_png my_png)` needs to behave as a pair of PNGs, despite appearing in a dynamically typed context. Type soundness alone only guarantees information according to the $*$ type of the application, so a semantics that "forgets" the cast injected by the translation around `segment_png` in the untyped context would still be sound.

This last point shows exactly why the syntactic tag soundness guarantee for Transient is not sufficient to explain what type-level reasoning power Transient offers. As discussed above, in Transient the avoidance of proxies leads to tag soundness. However, the Forgetful [Greenberg 2015] and Amnesic [Greenman et al. 2019a] variants of the Natural semantics do create the same proxy as Natural but then remove them when proxied values are used in dynamically typed contexts to reduce the running time and memory cost from proxies. In terms of the example, `segment_png` obtains the expected proxy, which is then removed when the function is used in `foreground`. However, Forgetful and Amnesic are as syntactically type sound as Natural, even though their net effect in terms of checks is the same as that of Transient. From the perspective of syntactic type soundness, in dynamically typed code, it does not matter whether the type of `segment_png` is enforced. That is, we must reason beyond syntactic type soundness to ensure types are meaningful.

Unfortunately, semantic type soundness, the next step up from syntactic type soundness, has a similar deficiency. Semantic type soundness asks that a value behaves according to its *latest* type, the one the current context expects. Hence, it also ignores intermediate types.

2.4 Complete Monitoring is Not Enough

An attempt to deal with the issue of intermediate unchecked types is complete monitoring for gradual types [Greenman et al. 2023, 2019a], which adapts the notion of complete monitoring from work on contract systems [Dimoulas et al. 2012]. The starting point for complete monitoring is a collection of semantics for an ICTL, i.e., different semantics with the same syntax and the same simple type system. The goal is to determine which of the semantics enforces the types of ICTL programs completely. However, complete monitoring establishes a weaker property. Intuitively, an ICTL semantics is a complete monitor if it "has complete control over every type-induced channel of communication between two components." [Greenman et al. 2023].

Formally, complete monitoring relies on a brittle notion of ownership of program expressions and values by components. In detail, in the complete-monitoring framework, components are encoded as label annotations on expressions; all expressions that "belong" to the same component have the label of the component. A system of axioms determines how values may accumulate labels, (and therefore component-owners), as they 'flow" from one component to another during evaluation. Another set of axioms describes how values lose labels due to checks from type casts. Given this formal setup, complete monitoring becomes preservation of a single-ownership invariant for all values during the evaluation of a program: a value can either have a single label, or multiple that are separated by type casts. Hence, the single-ownership property entails that the ICTL semantics is "in control of" all flows of values from one component to another as it imposes checks that regulate such flows. In that sense, the Natural semantics is a complete monitor, but Transient is not.

Back to the scenario in Fig. 2, since the Natural semantics is a complete monitor, every flow is checked. Assuming that `segment` and `segment_png` have labels l_1 and l_2 respectively, as the first flows into the second, `segment_png` becomes `segment_png` = $(\texttt{cast } \{\texttt{PNG -> PNG} \times \texttt{PNG}\} \texttt{ segment}_{l_1})_{l_2}$. When `segment_png` is applied in a dynamically typed context with label l_3, the call to `segment_png` becomes $(\texttt{cast } \{ \texttt{ * } \} \texttt{ segment\_png}_{l_2})_{l_3}$. While the expression has multiple labels, they are separated by a proxy, and hence, the semantics maintains the single-owner policy. Furthermore, when `segment_png` is applied, checks discharge the ownership labels for the result of the function and allow it to obtain the label of its calling component.

Note though, that the description above is intentionally vague about what checks exactly have to happen at each point in the evaluation where complete monitoring prescribes a check. This is because the formal framework of complete monitoring does not specify that much: an ICTL semantics would still be a complete monitor if all checks were equivalent to a no-op. Hence, a language designer cannot reason about what types a semantics enforces based solely on complete monitoring.

In summary, while complete monitoring offers a dimension that type soundness lacks, i.e., that checks should mediate the interactions of components, it is not a solution to determining whether a translation and ICTL semantics combination is adequate for the GTL type system. First, it is coarse-grained; it only gives a negative answer for any other semantics except Natural. Importantly, it cannot shed light on the guarantees the Transient approach offers. Second, it requires defining a brittle syntactic system of annotations, along with an instrumented semantics that needs to be adapted in an ad hoc manner from one approach to another. Finally, it entails a rather weak relation between the type system of the ICTL and its semantics.

2.5 Enter Vigilance

Intuitively, a GTL should come with reliable type enforcement, so that types can be used for reasoning. Returning to the example in Fig. 2, under either the Natural or Transient approach, the result of `segment_png my_png` should live up to its return type of pairs of PNGs.

To capture this intuition, this paper develops a new semantic property called *vigilance*. The goal of vigilance is to describe how the combination of the translation and ICTL semantics impacts the enforcement of the types in a GTL program. Vigilance serves to identify when the behavior of translated GTL programs are not a good match for the expectations implied by the statics of the GTL, and goes beyond (semantic) type soundness and complete monitoring. By requiring that the translation and semantics enforce every type obligation a value acquires during the evaluation of a program, **vigilance considers flow-sensitive and compositional type information that type soundness ignores**. And, instead of only specifying the location of checks as in complete monitoring, **vigilance ensures and allows for exactly enough dynamic enforcement to enforce the type-based properties of programs**. And by varying the type system, and the combination of translation and semantics, and thereby varying the typing histories for expressions and their enforcement, **vigilance is applicable to more systems than complete monitoring**.

When a translation and semantics is not vigilant for a type system, then there are some types that the type system requires, but the semantics is insufficient to ensure. In this case, a language designer can weaken the type system exactly where the translation and semantics is insufficient. And when a translation and semantics is vigilant for a type system, then there may be some stronger type system that the translation and semantics will still enforce. By providing these signals to language designers, **vigilance acts as a compass to guide language design**. In §2.7 and §2.8, we use vigilance as a compass for Transient and demonstrate an alternative design point.

Vigilance: Preliminaries. While the discussion so far centers around the simple type system, our goal is to pinpoint what combinations of translations and ICTL semantics are (in)adequate for what GTL type systems. For that, we need to overcome two technical challenges. First, our framework needs to accommodate different GTL type systems and connect the types of GTL expressions under each type system with the types of the ICTL values they produce when run. Second, the framework needs to keep track of the types ICTL values collect when run.

For the first challenge, for each type system that we wish to consider, vigilance asks for a **type-preserving translation** that maps from well-typed GTL programs to well-typed ICTL programs. Since the translation is type preserving, we reduce the relationship between the translation, ICTL semantics and GTL types to the relationship between ICTL semantics and ICTL types.

For the second challenge, vigilance relies on naming values in order to associate with each value a list of types it must satisfy, so we employ an allocating ICTL operational semantics. Specifically, we allocate every value v that arises during evaluation (including the results of casts and assertions) to a fresh label ℓ in a *value log* Σ, and modify elimination forms to act on labels ℓ to eliminate the value associated with the label $\Sigma(\ell)$. On every cast that evaluates to a label, the type from the cast is also stored. Unlike complete monitoring, the allocating semantics in our framework are

just a system for naming values and tracking types over a predefined semantics, which is mostly mechanical. Moreover, while one can use the type information associated with every intermediate value to perform checks (as we do in the semantics defined in §3), this is by no means necessary for vigilance, and one can have a completely distinct enforcement mechanism. In other words, the collected type information is effectively *ghost state* (originally *auxiliary variables* [Owicki and Gries 1976]), and the allocating semantics maintains a trivial erasure into the non-allocating one.

Vigilance: Technical Definition. After equipping our ICTL with an allocating semantics, we define vigilance using a (step-indexed) unary logical relation that models ICTL types τ as sets of ICTL values v that inhabit them. In contrast to a typical step-indexed logical relation for type soundness [Appel and McAllester 2001], vigilance comes with an extra index, the *typing history* Ψ, which can be thought of as a value-log typing. Specifically, the typing history Ψ is a log that collects the types on each cast or assertion in the program that a particular value has passed through. While semantic type soundness says that a language is semantically type sound if and only if any well-typed expression $e : \tau$ "behaves like" τ, vigilance asks that e also "behaves" according to its typing history. We say that the semantics of an ICTL are *vigilant* for a type system if in any well-formed typing history Ψ (capturing potential casts and assertions from a context), the translation of e behaves like τ. The latter means that, if evaluating e produces a label ℓ (as well as a potentially larger typing history Ψ', capturing casts and assertions present in e), then $\Sigma(\ell)$ not only behaves like τ (in the conventional sense), but also like all of the types in $\Psi'(\ell)$.

In summary, an ICTL semantics is vigilant for an ICTL type system and GTL translation when any well-typed GTL expression t at type τ translates to an ICTL expression e that behaves like τ according to the vigilance logical relation.

Vigilance: a Recipe. The above discussion implies a step-by-step recipe for applying vigilance to a gradual typing approach:

(1) Define a GTL, an ICTL and their type systems.
(2) Define a type-preserving translation from the GTL to the ICTL.
(3) Define an allocating semantics for the ICTL.
(4) Define the vigilance logical relation for the ICTL.
(5) Attempt to prove the fundamental property of the vigilance relation.
(6) If the proof fails, retry after adjusting the GTL and ICTL type systems, the type-preserving relation, or the checks that the semantics of the ICTL performs. Changes to the type system may require an adjustment to the vigilance relation so that it reflects the semantics of types that correspond to the adjusted type system.

Constructing the vigilance relation is the most involved step of the recipe. However, the vigilance relation extends a standard type soundness relation, and the required extensions for the typing history for a new linguistic setting only asks for similar "semantic thinking" as that needed for defining the soundness relation itself. We provide a high-level discussion of typing histories through an example in the remainder of this section, and the full formal details for how they can be incorporated in a soundness logical relation is in §4. The remainder of the section, and §5 formally, also demonstrate two different ways the pieces of our framework can be adjusted when a first attempt to prove the fundamental property for the vigilance relation fails.

As a final remark herein, a contributing factor to the complexity of the formal development in this paper is not due to vigilance itself. Instead, the complexity comes from the fact that we use vigilance as a tool for comparing Natural and Transient. In particular, as we discuss in §2.1, in order to make a meaningful, apples-to-apples, comparison we have to carefully craft our GTL and ICTL so that they can support both Natural and Transient while eliminating their superficial differences.

If a comparison is not the goal, one can avoid the design of a unified framework and simply focus on the steps of the vigilance recipe.

2.6 Vigilance: By Example

A concrete discussion about vigilance requires an illustration of the contents of the typing history of a value. To that end, we analyze the evaluation of the example in Fig. 3. The example condenses the scenario in Fig. 2 as a single-component program in our GTL. In particular, just as in the scenario, the `crop_png` function is a typed wrapper around the dynamically typed `crop` function, which the user code applies to the image `my_png`. The resulting cropped image is written to file `"my.png"`. Different from previous examples and for presentation purposes only, the example comes with partial label annotations. Specifically, we write $(e)_l$ to denote an expression e annotated with a label l that uniquely identifies the value e evaluates to. In other words, the labels are a presentation device that acts as a layer of indirection so that we can refer to program values and relate them with types.

```
let crop = λ img. (...)
let crop_png = λ(img:PNG) → PNG. (crop img)_l1
write_img (crop_png (my_png)_l0)_l2 "my.png"
```

Fig. 3. Example to Illustrate Labels and Typing History

Analyzing the evaluation of the example from the perspective of typing history justifies why writing the cropped version of `my_png` to the file as an image is a reasonable choice. In particular, consider the typing histories of `my_png` before, during and after the call to `crop_png`, which correspond to labels l_0, l_1 and l_2 respectively. The image originates in dynamically typed code and has label l_0. That label is associated with just the type obligation $*$ in the typing history of the value. When the value flows through `crop_png`, its new label l_1 accumulates the obligation PNG on top of the obligation $*$ it inherits from l_0. Finally, after cropping, the image returns to dynamically typed code with label l_2, which is associated with the latest obligation $*$ in addition to obligation PNG and $*$ from l_1. As a result, if the statics and dynamics of the language enforce the typing history for l_2, the image meets the obligation PNG, and therefore can be safely written to the file as an image without any format checks.

It is worth noting that the above discussion does not depend on how the semantics of the ICTL performs checks, but only on whether the typing history of l_2 is enforced. Consequently, any combination of statics, translation and dynamics that enforces the typing history enables the described typed-based reasoning. Vigilance generalizes this point: when a translation-semantics combination is vigilant for a type system, every value produced during the evaluation of a program satisfies its typing history, and hence code that uses a value can safely assume that much. For example, the example in Fig. 2 shows how the simple type system, the Uni translation and the Natural semantics work together to enforce the typing history of all values; the calls to `segment_png` behave not only according to type $*$, but also, according to type $\mathsf{PNG} \to \mathsf{PNG} \times \mathsf{PNG}$. In contrast, the same is not true for the simple type system, the Uni translation and the Transient semantics.

2.7 Vigilance: An Examination of Transient

Since the combination of the Uni translation and the Transient semantics is not vigilant for the standard simple type system, we investigate: *for what type system is the combination of* Uni *and* Transient *vigilant?* An initial answer, which confirms prior work on Transient and tag soundness, is that one such type system ascribes type tags (top-level type constructors) to expressions rather than types. Hence, this *tag type system* accepts programs that have imprecise types, which shows the difference between what the simple type system promises and what the Uni translation and Transient semantics seems to achieve.

However, the Transient approach has deeper type-level reasoning than tag soundness suggests. The typing history, which is central to vigilance, makes this additional power plain. Consider again

the scenario in Fig. 2. As described before, the scenario uses a dynamically typed image library that provides two functions: `crop`, which crops images to a particular size, and `segment` which segments an image into a pair of a "foreground" and a "background" image. The typed image library provides three typed wrappers of these functions as `segment_png`, `segment_png_small` and `crop_png`. However, in the case of `segment_png_small`, the typed wrapper does a bit more than just acting as a veneer of types; it uses `crop_png` to reduce the sizes of the pair of images that `segment` produces. Since the Uni translation and Transient semantics result in tag checks for pair projections, the Transient approach should guarantee the return type of `segment_png_small`, not just $*\times*$. This conclusion is a direct consequence of the fact that the combination of Uni and Transient is vigilant for the tag type system; vigilance entails that Transient checks that the results of the calls to `crop_png` in the body of `segment_png_small` are PNGs. Hence, one can deduce the result of `segment_png_small` has indeed type PNG×PNG. In conclusion, there should be a translation, such that its combinations with Transient should be vigilant for a stronger type system than the tag one, namely one which recognizes `segment_png_small` has return type PNG×PNG.

2.8 Vigilance: Towards Truer Transient Types

Building on the above insight, as a design exercise, we use vigilance as a guide towards a revised type system and type-preserving translation. The new *truer* type system makes limited use of union and intersection types in order to reflect in the types of expressions the outcomes of Transient casts and assertions injected by the new Flow translation.

Importantly, the truer type system showcases how vigilance can benefit language designers. They can start from an adequate design point, such as the variant of the Transient approach with a tag type system, and, with vigilance as a guide, find others. If a new adequate design point involves a more precise type system, such as the truer one, the designer may use it as part of IDE and refactoring tools, or for optimizing the dynamics of the language. For instance, a consequence of truer is that some of the checks that are necessary so that the translation and Transient semantics is vigilant for the tag type system can be elided in a provably correct manner when pairing the Flow translation and Transient semantics with the truer type system. For example, the truer type system can stitch together type information from the type assertions on the results of the calls to `png_crop` in the body of `segment_png_small` to deduce statically that `segment_png_small` indeed has a type that precisely matches its type annotation (rather than that it simply returns a pair that should be checked further at run time). Moreover, this precise truer type makes unnecessary a type assertion on the outcome of the left-pair projection that gets bound to `foreground`; it is statically known that it is a PNG. To formally establish the above, in §5.4 we use the truer type system to eliminate unnecessary transient checks in a semantics- and vigilance-preserving manner.

2.9 Technical Contributions

The main technical contribution of this paper is vigilance as an analytical tool for gradual typing. The table on the right summarizes results from our comparative analysis of the well studied Natural and Transient approaches. When the cell contains a ✓, that combination of translation and semantics is vigilant for that type system; otherwise it is not. While §1 presents our results in terms of the HO-Natural and FO-Transient combinations, as we discuss throughout §2, Uni consolidates HO and FO. In particular, when an expression is translated according to Uni, the HO can be recovered by erasing the type assertions that are specific to the Transient approach, which is exactly what the Natural ICTL semantics achieves by treating them as no-ops. Hence, the Uni-Natural and Uni-Transient correspond to the standard presentation of the Natural and Transient

	Simple	Tag
Uni-Natural	✓ (Thm. 4.1)	× (Thm. 5.6)
Uni-Transient	× (Thm. 4.3)	✓ (Thm. 5.4)

$$
\begin{array}{rcl}
t & ::= & x \mid n \mid i \mid \mathsf{True} \mid \mathsf{False} \mid \lambda(x{:}\tau) \to \tau'.\, t \mid \langle t, t \rangle \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \\
 & \mid & \mathit{binop}\, t\, t \mid t\ t \mid \mathsf{fst}\, t \mid \mathsf{snd}\, t \\
\tau & ::= & \mathsf{Nat} \mid \mathsf{Int} \mid \mathsf{Bool} \mid \tau \times \tau \mid \tau \to \tau \mid * \\
\\
e & ::= & x \mid n \mid i \mid \mathsf{True} \mid \mathsf{False} \mid \lambda(x{:}\tau).\, e \mid \langle e, e \rangle \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
 & \mid & \mathit{binop}\, e\, e \mid \mathsf{app}\{\tau\}\, e\, e \mid \mathsf{fst}\{\tau\}\, e \mid \mathsf{snd}\{\tau\}\, e \mid \mathsf{cast}\,\{\tau \Leftarrow \tau\}\, e
\end{array}
\qquad
\begin{array}{rcl}
\mathit{binop} & ::= & \mathsf{sum} \mid \mathsf{quotient} \\
n \in \mathbb{N} & , & i \in \mathbb{Z}
\end{array}
$$

Fig. 4. Syntax of λ^{GTL} (top) and λ^{ICTL} (bottom)

approaches respectively in the literature. Importantly, and distinctly from the literature, the four results establish that tag typing is not strictly semantically weaker than the simple type system; tag typing allows a function to be applied in more contexts, which is well behaved when under the Uni and Transient combination, but not under the Uni-Natural combination. Transient's checks protect the function from these contexts while Natural's do not. Hence, vigilance characterizes the Transient approach positively in a way that distinguishes it from the Natural approach.

Besides comparing existing points in the design space of gradual typing, the paper also shows how vigilance can guide the search for new points. In this spirit, the final result of the paper, which is not shown in the table, takes the positive characterization of the Transient approach one step further. It demonstrates that the Flow-Transient combination is vigilant for the truer type system (Thm. 5.10), which enables the elision of unnecessary dynamic checks (Thm. 5.12).

3 FROM A GTL TO AN ICTL WITH TWO SEMANTICS

The top portion of Fig. 4 presents the syntax of λ^{GTL}, our GTL, which as described in §2 follows the approach of Greenman and Felleisen [2018] and the gradually-typed λ-calculus [Siek and Taha 2006]. Most of the features of λ^{GTL} are the same as the corresponding features of a simply-typed λ-calculus extended with constants, pairs and their relevant elimination forms. The one unconventional syntactic form is that for anonymous functions. In particular, anonymous functions come with type annotations that describe both the type of their arguments and the type of their result. The type annotations τ range over *simple types* with the addition of $*$, the dynamic type, which, as usual in the gradual typing setting, indicates imprecise or missing type information. For example, the expression $\lambda(x : * \to *) \to *.\, t$ represents an anonymous function that consumes functions and may return anything.

Since λ^{GTL} expressions t do not evaluate directly but are translated to an ICTL, before delving into the type checking and evaluation of λ^{GTL} expressions, we discuss briefly the syntax of λ^{ICTL}, our ICTL. The bottom portion of Fig. 4 shows the syntax of λ^{ICTL} expressions e. Its features correspond to those of λ^{GTL} with a few important differences. First, functions $\lambda(x{:}\tau).\, e$ come with type annotations for their arguments but not their results. Second, pair projections and function applications also have type annotations. Third, λ^{ICTL} has a new syntactic form compared to λ^{GTL}: *cast* expressions. Specifically, $\mathsf{cast}\,\{\tau_1 \Leftarrow \tau_2\}\, e$ represents a cast from type τ_2 to τ_1 for the result of expression e. In other words, while in λ^{GTL} all type annotations are on functions, in λ^{ICTL}, they are spread over applications, pair projections, function parameters, and casts. This is because the first three are the syntactic loci in a program that correspond to "boundaries" between pieces of code that can have types with different precision according to the type system of the GTL. Hence, the translation injects type assertions and casts exactly at these spots. The type annotations correspond to the type assertions that the Transient approach relies on to make up for the weak checks performed by Transient casts (the notation and separation of assertions from casts comes from Greenman and Felleisen [2018]). After all, as we discuss in §2, λ^{ICTL} aims to accommodate both Natural and Transient in a uniform linguistic setting.

$\boxed{\Gamma \vdash_{\mathsf{Uni}} t : \tau \rightsquigarrow e}$ (Unified Translation, selected rules)

$$\frac{\Gamma,\ (x{:}\tau) \vdash_{\mathsf{Uni}} t : \tau'' \rightsquigarrow e}{\Gamma \vdash_{\mathsf{Uni}} \lambda(x{:}\tau) \to \tau'.\, t : \tau \to \tau' \rightsquigarrow \lambda(x{:}\tau).\, ([\tau' \swarrow \tau'']e)}$$

$$\frac{\Gamma \vdash_{\mathsf{Uni}} t_1 : \tau \to \tau' \rightsquigarrow e_1 \qquad \Gamma \vdash_{\mathsf{Uni}} t_2 : \tau'' \rightsquigarrow e_2}{\Gamma \vdash_{\mathsf{Uni}} t_1\ t_2 : \tau' \rightsquigarrow \mathsf{app}\{\tau'\}\ e_1\ ([\tau \swarrow \tau'']e_2)}$$

$$[\tau \swarrow \tau']e = \begin{cases} e & \text{if } \tau' \leqslant: \tau \\ \mathsf{cast}\,\{\tau \Leftarrow \tau'\}\ e & \text{if } \tau' \not\leqslant: \tau \\ & \text{and } \tau \sim \tau' \end{cases}$$

$$\frac{\Gamma \vdash_{\mathsf{Uni}} t_1 : * \rightsquigarrow e_1 \qquad \Gamma \vdash_{\mathsf{Uni}} t_2 : \tau' \rightsquigarrow e_2}{\Gamma \vdash_{\mathsf{Uni}} t_1\ t_2 : * \rightsquigarrow \mathsf{app}\{*\}\ (\mathsf{cast}\,\{* \to * \Leftarrow *\}\ e_1)\ ([* \swarrow \tau']e_2)}$$

$$\frac{\Gamma \vdash_{\mathsf{Uni}} t_b : \mathsf{Bool} \rightsquigarrow e_b \qquad \Gamma \vdash_{\mathsf{Uni}} t_1 : \tau_1 \rightsquigarrow e_1 \qquad \Gamma \vdash_{\mathsf{Uni}} t_2 : \tau_2 \rightsquigarrow e_2}{\Gamma \vdash_{\mathsf{Uni}} \mathsf{if}\ t_b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \tau_1 \mathbin{\widetilde{\sqcup}} \tau_2 \rightsquigarrow \mathsf{if}\ e_b\ \mathsf{then}\ ([\tau_1 \mathbin{\widetilde{\sqcup}} \tau_2 \swarrow \tau_1]e_1)\ \mathsf{else}\ ([\tau_1 \mathbin{\widetilde{\sqcup}} \tau_2 \swarrow \tau_2]e_2)}$$

$\boxed{\tau \sim \tau'}$ (Compatibility)

$$\frac{}{\tau \sim *} \qquad \frac{}{\mathsf{Nat} \sim \mathsf{Int}} \qquad \frac{\tau_0 \sim \tau_2 \qquad \tau_1 \sim \tau_3}{\tau_0 \times \tau_1 \sim \tau_2 \times \tau_3} \qquad \frac{\tau_0 \sim \tau_2 \qquad \tau_1 \sim \tau_3}{\tau_0 \to \tau_1 \sim \tau_2 \to \tau_3} \qquad \frac{}{\tau \sim \tau} \qquad \frac{\tau \sim \tau'}{\tau' \sim \tau}$$

$\boxed{\tau \leqslant: \tau'}$ (Subtyping)

$$\frac{}{\mathsf{Nat} \leqslant: \mathsf{Int}} \qquad \frac{\tau_0 \leqslant: \tau_2 \qquad \tau_1 \leqslant: \tau_3}{\tau_0 \times \tau_1 \leqslant: \tau_2 \times \tau_3} \qquad \frac{\tau_2 \leqslant: \tau_0 \qquad \tau_1 \leqslant: \tau_3}{\tau_0 \to \tau_1 \leqslant: \tau_2 \to \tau_3} \qquad \frac{}{\tau_0 \leqslant: \tau_0}$$

Fig. 5. The Unified Uni Translation From λ^{GTL} to λ^{ICTL}

Fig. 5 presents the Uni translation of λ^{GTL} expressions t to λ^{ICTL} expressions e with a single judgment $\Gamma \vdash_{\mathsf{Uni}} t : \tau \rightsquigarrow e$. A λ^{GTL} function translates at its type annotation $\tau \to \tau'$ if its body translates at some type τ''. To bridge the potential gap between τ'' and τ', the translation of the function produces a λ^{ICTL} function whose body is wrapped in a cast from τ'' to τ', if needed. Specifically, metafunction $[\tau \swarrow \tau']e$ inserts a cast around e when τ is *compatible* with τ' (written $\tau \sim \tau'$) but not a subtype of τ'. The metafunction $[\tau \swarrow \tau']e$ is designed so that subtyping allows *implicit* (no cast) type conversion, while compatibility allows *explicit* (casted) type conversion. Subtyping is defined in the canonical way: Nat is a subtype of Int, functions are contravariant, and pairs are covariant. Compatibility is the reflexive and symmetric relation that rules out non-convertible type casts, or type casts that will always error. Unlike standard definitions, compatibility includes $\mathsf{Nat} \sim \mathsf{Int}$ to allow programmers to freely convert between Naturals and Integers, and have the translation insert appropriate checks. Our compatibility is a symmetric version of consistent subtyping [Bañados Schwerter et al. 2021][6].

Conditionals translate in a recursive manner. The type of the conditional is the consistent subtype join $\widetilde{\sqcup}$ of the types of its two branches. The consistent subtype join definition is standard [Bañados Schwerter et al. 2021], and gives the least upper bound of the types with respect to subtyping, as well as more precise types in place of $*$. To bridge the potential gap between the type of the branch and the consistent join, the translation may wrap each branch in a cast with the same metafunction as above. Translated applications obtain type assertions for the return type of the applied function, along with (possible) casts around the argument expression that make sure the domain of the applied function jives with the type of the provided argument.

[6]If we used compatible subtyping here, a conversion from Int to Nat would require calling a function with return type $*$. By splitting consistent subtyping into a compatibility relation and a subtyping relation, the type system accepts more programs and simplifies the translation — the definition of $[\tau \swarrow \tau']e$ becomes straightforward.

$\boxed{\Gamma \vdash_{\mathsf{sim}} e : \tau}$ (selected rules)

$$\frac{\Gamma_0 \vdash_{\mathsf{sim}} e_0 : \tau_0 \to \tau_1 \qquad \Gamma_0 \vdash_{\mathsf{sim}} e_1 : \tau_0}{\Gamma_0 \vdash_{\mathsf{sim}} \mathsf{app}\{\tau_1\}\, e_0\, e_1 : \tau_1} \qquad \frac{\Gamma_0 \vdash_{\mathsf{sim}} e_0 : \tau_0}{\Gamma_0 \vdash_{\mathsf{sim}} \mathsf{cast}\,\{\tau_1 \Leftarrow \tau_0\}\, e_0 : \tau_1} \qquad \frac{\Gamma_0 \vdash_{\mathsf{sim}} e_0 : \tau_0 \qquad \tau_0 \leqslant: \tau_1}{\Gamma_0 \vdash_{\mathsf{sim}} e_0 : \tau_1}$$

Fig. 6. Simple Typing for λ^{ICTL}

Fig. 6 gives rules for the typing judgment for λ^{ICTL}: $\Gamma \vdash_{\mathsf{sim}} e : \tau$ — the sim annotation indicates the standard simply typed ICTL type system, to distinguish it from the tag and tru systems we present later. In general, the type system of λ^{ICTL} is straightforward and closely follows the translation of λ^{GTL}. The translation has a key property: it maps well-typed λ^{GTL} expressions to well-typed λ^{ICTL} expressions with the same type.

Theorem 3.1 (Uni is Type-Preserving). *If* $\Gamma \vdash_{\mathsf{Uni}} t : \tau \rightsquigarrow e$ *then* $\Gamma \vdash_{\mathsf{sim}} e : \tau$.

ICTL Type Guarantees Lift to the GTL. As discussed in §2, a type-preserving translation, such as Uni, allows us to focus on the semantics of λ^{ICTL} to analyze enforcement of the types of a well typed λ^{GTL} expression t. Since the semantics of an λ^{GTL} expression t of type τ is defined by translation into a λ^{ICTL} expression e also of type τ, the typing history of values produced by the evaluation of t is enforced only if the the typing history of values produced by the evaluation of e is enforced. Therefore, the question of whether a translation-semantics combination for λ^{ICTL} is vigilant for its static type system reduces to whether the semantics of λ^{GTL} enforce the typing histories of the values produced during the evaluation of well-typed λ^{ICTL} expressions.
Note: The complete formal development of λ^{GTL} and λ^{ICTL} along with all the definitions, theorems and proofs are in the supplemental material.

3.1 A Natural and a Transient Semantics for λ^{ICTL}

The definition of vigilance, which is the centerpiece of this paper, requires an apparatus for determining the types associated with the value of each (sub)expression in a program — intuitively, all the types in casts applied to that value — so that vigilance can decide if the semantics of the ICTL indeed enforces these types. Such an apparatus needs to be dynamic in order to be precise in a higher-order gradually typed setting, such as λ^{ICTL}. Consider, for instance, the expression $e_1 = \mathsf{if}\ e_b\ \mathsf{then}\ \mathsf{cast}\,\{* \Leftarrow \tau\}\ \mathsf{cast}\,\{\tau \Leftarrow *\}\ e_0\ \mathsf{else}\ e_0$. If the result of e_0 is a value v_0, then depending on the result of e_b, v_0 is associated with different types: if e_b evaluates to True, then v_0 is associated with τ, and otherwise it is not.

To record these types, we devise a *log-based* reduction semantics for λ^{ICTL}. This semantics creates fresh labels ℓ for each intermediate value during the evaluation of a program to distinguish between different values that are structurally the same, and then uses the labels to track the (two) types from any casts that a label encounters during the evaluation of a program. Formally, the dynamic semantics maintains a *value log* Σ, which is a map from labels ℓ to values v and potential types $option(\tau \times \tau)$. The type information is optional because a value may never go through a cast.

The definition of the *log-based* reduction semantics requires an extension of the syntax of λ^{ICTL} with values, labels, unannotated applications, errors and, most importantly, expressions that correspond to the run-time representations of type casts and assertions. Essentially, these act as hooks that allow us to define either a Natural or a Transient semantics for λ^{ICTL} while leaving the rest of the formalism unchanged. The top left of Fig. 7 depicts these extensions. The *monitor* expression $\mathsf{mon}\,\{\tau \Leftarrow \tau\}\, e$ regulates the evaluation of cast expressions; it is an intermediate expression that separates the tag checks performed by a cast from the creation of a proxy. An assert expression, $\mathsf{assert}\ \tau\ e$, reifies type annotations on applications and function parameters as type assertions. Unannotated applications correspond to applications whose annotation has been reified as a type assertion.

$$
\begin{array}{lcl}
v & ::= & \ell \mid n \mid i \mid \mathsf{True} \mid \mathsf{False} \mid \langle \ell, \ell \rangle \mid \lambda(x:\tau).\, e \\
e & ::= & \ldots \mid \ell \mid e\ e \mid \mathsf{mon}\,\{\tau \Leftarrow \tau\}\, e \mid \mathsf{assert}\,\tau\, e \mid \mathsf{Err} \\
\Sigma & : & \mathbb{L} \to \mathbb{V} \times \mathsf{option}(\mathbb{T} \times \mathbb{T})
\end{array}
$$

$$\boxed{\mathsf{pointsto}(\Sigma, \ell)}$$

$$\mathsf{pointsto}(\Sigma, \ell) = \begin{cases} fst(\Sigma(\ell)) & \text{if } fst(\Sigma(\ell)) \neq \ell' \\ \mathsf{pointsto}(\Sigma, \ell') & \text{if } fst(\Sigma(\ell)) = \ell' \end{cases}$$

$\boxed{\Sigma, e \hookrightarrow_L \Sigma, e}$ (selected rules)

$$
\begin{array}{l}
\Sigma, v \\
\hookrightarrow_L \Sigma[\ell \mapsto (v, \mathsf{none})], \ell \\
\quad \text{where } \ell \notin dom(\Sigma)
\end{array}
$$

$$
\begin{array}{l}
\Sigma, \mathsf{app}\{\tau_0\}\ \ell_0\ \ell_1 \\
\hookrightarrow_L \Sigma, \mathsf{assert}\,\tau_0\ (\ell_0\ \ell_1)
\end{array}
$$

$$
\begin{array}{l}
\Sigma, \ell_0\ \ell_1 \\
\hookrightarrow_L \Sigma, e_0[x_0 \leftarrow \ell_1] \\
\quad \text{if } \Sigma(\ell_0) = (\lambda(x_0:\tau_1).\, e_0, \_) \\
\quad \text{and } \mathsf{pointsto}(\Sigma, \ell_1) \propto^L_{assert} \tau_1
\end{array}
$$

$$
\begin{array}{l}
\Sigma, \ell_0\ \ell_1 \\
\hookrightarrow_L \quad \mathsf{mon}\,\{cod(\tau_1) \Leftarrow cod(\tau_2)\} \\
\qquad\quad (\ell_2\ \mathsf{mon}\,\{dom(\tau_2) \\
\qquad\qquad\quad \Leftarrow dom(\tau_1)\}\ \ell_1) \\
\quad \text{if } \Sigma(\ell_0) = (\ell_2, \mathsf{some}(\tau_1, \tau_2))
\end{array}
$$

$$
\begin{array}{l}
\Sigma, \mathsf{assert}\,\tau_0\ \ell_0 \\
\hookrightarrow_L \Sigma, \ell_0 \\
\quad \text{if } \mathsf{pointsto}(\Sigma, \ell_0) \propto^L_{assert} \tau_0
\end{array}
$$

$$
\begin{array}{l}
\Sigma, \mathsf{assert}\,\tau_0\ \ell_0 \\
\hookrightarrow_L \Sigma, \mathsf{TypeErr}(\tau_0,\ \ell_0) \\
\quad \text{if } \neg\mathsf{pointsto}(\Sigma, \ell_0) \propto^L_{assert} \tau_0
\end{array}
$$

$$
\begin{array}{l}
\Sigma, \mathsf{cast}\,\{\tau_1 \Leftarrow \tau_2\}\ \ell_0 \\
\hookrightarrow_L \Sigma, \mathsf{mon}\,\{\tau_1 \Leftarrow \tau_2\}\ \ell_0 \\
\quad \text{if } \mathsf{pointsto}(\Sigma, \ell_0) \propto^L_{cast} \tau_1 \\
\quad \text{and } \mathsf{pointsto}(\Sigma, \ell_0) \propto^L_{cast} \tau_2
\end{array}
$$

$$
\begin{array}{l}
\Sigma, \ell_0\ \ell_1 \\
\hookrightarrow_L \Sigma, \mathsf{Wrong} \\
\quad \text{if } \Sigma(\ell_0) = (v, \_) \\
\quad \text{and } v \notin \lambda(x:\tau).\, e \cup \ell \\
\quad \text{or } \Sigma(\ell_0) = (\ell_0', \mathsf{none})
\end{array}
$$

$\boxed{\propto: v \times K \longrightarrow \mathbb{B}}$

$n \propto \mathsf{Nat}$	$= \mathsf{True}$
$i \propto \mathsf{Int}$	$= \mathsf{True}$
$b \propto \mathsf{Bool}$	$= \mathsf{True}$
$\langle v_1, v_2 \rangle \propto * \times *$	$= \mathsf{True}$
$(\lambda(x:\tau).\, e) \propto * \to *$	$= \mathsf{True}$
$(\mathsf{mon}\,\{\tau \Leftarrow \tau'\}\, v) \propto * \to *$	$= \mathsf{True}$
$v \propto *$	$= \mathsf{True}$
$v \propto K$	$= \mathsf{False}$
	otherwise

$\boxed{\propto^L_c: v \times K \longrightarrow \mathbb{B}}$

c	$v \propto^{\mathsf{Natural}}_c \tau$	$v \propto^{\mathsf{Transient}}_c \tau$
cast	$v \propto \lfloor \tau \rfloor$	$v \propto \lfloor \tau \rfloor$
mon	$v \propto \lfloor \tau \rfloor$	True
assert	True	$v \propto \lfloor \tau \rfloor$

Fig. 7. Evaluation Syntax and Reduction Semantics for λ^{ICTL}

There are two kinds of errors in the evaluation language of λ^{ICTL}: $\mathsf{Err}^\bullet$ are expected errors and include failures due to failed type casts and assertions, and Err° are unexpected errors that indicate a failure of type soundness such as a call to a value that is not a function. Err ranges over these.

The two semantics of λ^{ICTL} are defined with the reduction relation $\longrightarrow^*_L$ that is the transitive, compatible closure (over evaluation contexts) of the notions of reductions $\hookrightarrow_L$, where L is either Natural or Transient. The only difference between the two notions of reduction is in their *compatibility* metafunction $\propto^L_c$, where the parameter c represents the kind of check being performed by the semantics. The metafunction consumes a value and a type, and either immediately returns True or invokes $v \propto \lfloor \tau \rfloor$ that checks whether v matches the *tag* $\lfloor \tau \rfloor$ of the given type τ. Put differently, $v \propto^L_c \tau$ either performs a tag check or is a no-op — which of the two depends on its c and L parameters, that is, the λ^{ICTL} construct that triggers a possible tag check and the particular semantics of λ^{ICTL}. Specifically, in both semantics, a cast expression performs a tag check. However, assert expressions perform tag checks only in Transient since in Natural all dynamic type checking takes place via proxies. Conversely, monitor expressions perform tag checks only in Natural since Transient does not rely on proxies for dynamic type checking.

The bottom part of Fig. 7 presents a few selected rules of $\hookrightarrow_L$. When an expression reduces a value, $\hookrightarrow_L$ replaces it with a fresh label ℓ and updates Σ accordingly. An annotated application becomes an unannotated one but wrapped in an assert expression that reifies the annotation as a type assertion. Unannotated applications delegate to the compatibility metafunction a potential check of the argument against the type of the parameter. When the compatibility metafunction returns True the evaluation proceeds with a beta-reduction; otherwise it raises a dynamic type error TypeErr. Since all values are stored in the value log and these rules need to inspect values, they employ metafunction $\mathsf{pointsto}(\cdot, \cdot)$. Given a value log Σ and a label ℓ, the metafunction traverses Σ starting from ℓ through labels that point to other labels until it reaches a non-label value. The case

$$\mathcal{V}^L_{\mathsf{sim}}[\![C]\!] \triangleq \{(k, \Psi, \Sigma, \ell) \mid \mathsf{pointsto}(\Sigma, \ell) \in C\}$$

$$\mathcal{V}^L_{\mathsf{sim}}[\![\tau_1 \times \tau_2]\!] \triangleq \{(k, \Psi, \Sigma, \ell) \mid \Sigma(\ell) = (\langle \ell_1, \ell_2 \rangle, \_) \wedge (k, \Psi, \Sigma, \ell_1) \in \mathcal{V}^L_{\mathsf{sim}}[\![\tau_1]\!] \wedge (k, \Psi, \Sigma, \ell_2) \in \mathcal{V}^L_{\mathsf{sim}}[\![\tau_2]\!]\}$$

$$\mathcal{V}^L_{\mathsf{sim}}[\![\tau_1 \to \tau_2]\!] \triangleq \{(k, \Psi, \Sigma, \ell) \mid \forall\ (j, \Psi') \sqsupseteq (k, \Psi),\ \Sigma' \supseteq \Sigma,\ \ell_v,\ \tau_0 \mathrel{:\!\!\geqslant} \tau_2.\ \Sigma' : (j, \Psi')\ \wedge (j, \Psi', \Sigma', \ell_v) \in \mathcal{V}^L_{\mathsf{sim}}[\![\tau_1]\!].\ \Rightarrow (j, \Psi', \Sigma', \mathsf{app}\{\tau_o\}\ \ell\ \ell_v) \in \mathcal{E}^L_{\mathsf{sim}}[\![\tau_0]\!]\}$$

$$\mathcal{V}^L_{\mathsf{sim}}[\![*]\!] \triangleq \{(k, \Psi, \Sigma, \ell) \mid (k-1, \Psi, \Sigma, \ell) \in \mathcal{V}^L_{\mathsf{sim}}[\![C]\!] \cup \mathcal{V}^L_{\mathsf{sim}}[\![* \times *]\!] \cup \mathcal{V}^L_{\mathsf{sim}}[\![* \to *]\!]\}$$

$$\mathcal{E}^L_{\mathsf{sim}}[\![\tau]\!] \triangleq \{(k, \Psi, \Sigma, e) \mid \forall j \leq k.\ \forall \Sigma' \supseteq \Sigma, e'.\ (\Sigma, e) \longrightarrow^j_L (\Sigma', e') \wedge \mathtt{irred}(e') \Rightarrow (e' = \mathsf{Err}^\bullet \vee (\exists (k-j, \Psi') \sqsupseteq (k, \Psi).\ \Sigma' : (k-j, \Psi') \wedge (k-j, \Psi', \Sigma', e') \in \mathcal{V}^L_{\mathsf{sim}}[\![\tau]\!]))\}$$

Fig. 8. Vigilance for Simple Typing: Value and Expression Relations

where an application does not involve a function is one of the cases that the type system of λ^{ICTL} should prevent. Hence, the reduction rule raises Wrong to distinguish this unexpected error.

Assert and cast expressions also delegate any tag checks they perform to the compatibility metafunction. If answer of the latter is positive, an assert expression simplifies to its label-body, while a cast expression wraps its value-body into a monitor with the same type annotations.

Monitor expressions essentially implement proxies, if the semantics of λ^{ICTL} relies on them. Specifically, a monitor expression performs any checks a proxy would perform using the compatibility metafunction, and produces a fresh label ℓ to record in the value log and associates ℓ with two additional types. Upon an application of a label, $\hookrightarrow_L$ retrieves the types associated with it, and creates a monitor expression to enforce them. Hence, if the compatibility metafunction does perform tag checks for monitor expressions, monitors implement the two steps of checking types with proxies: checking first-order properties of the monitored value, and creating further proxies upon the use of a higher-order value. If the compatibility metafunction does not perform tag checks then all these reduction rules are essentially void of computational significance; they are just a convenient way for keeping the semantics syntactically uniform across Natural and Transient.

An example sequence of reductions as well as a bisimulation theorem that includes the value equivalence relation is given in the supplemental material.

4 VIGILANCE, FORMALLY

In this section, we define *vigilance for simple typing*. It requires that every value produced during the evaluation of an ICTL expression must satisfy *both* the type ascribed to it by the simple type system *and* all the types from casts that were evaluated to produce this value. We refer to the latter list of types as the run-time typing history for the value. The first of these two conditions is essentially (semantic) type soundness which can be captured using a unary logical relation indexed by types and inhabited by values that satisfy the type. For the second condition, we must extend the logical relation to maintain a *type history* Ψ that keeps track of the run-time typing history h for each value v in the log Σ, and then require that each v satisfy all the types in its history h.

We start with the standard semantic-type-soundness part of our step-indexed logical relation. Fig. 8 presents the value and expression relations. Ignoring, for the moment, the highlighted terms in the figure, the value relation $\mathcal{V}^L_{\mathsf{sim}}[\![\tau]\!]$ specifies when a value stored at label ℓ in Σ satisfies the type τ for k steps — or, in more technical terms, when a Σ, ℓ pair belongs to τ. But each value relation is also indexed by a type history Ψ that, intuitively, records the run-time typing histories for all values in Σ, as we explain in detail later.

$$\vdash \Psi \triangleq \forall \ell.\, \Psi(\ell) = \tau, \tau', h \Rightarrow \tau' \coloneqq\!\geqslant \mathrm{head}(h)$$

$$\vdash \Sigma \triangleq \forall \ell \in dom(\Sigma).\, (\Sigma(\ell) = (v, \mathsf{none}) \wedge v \notin \mathbb{L})$$
$$\vee\, (\Sigma(\ell) = (\ell', \mathsf{some}(\tau', \tau)) \wedge \exists v.\, v = \mathsf{pointsto}(\Sigma, \ell) \wedge \neg(v \propto *\times *) \wedge v \propto \tau' \wedge v \propto \tau)$$

$$\Psi \vdash^{\ell} (v, \mathsf{none}) \triangleq \exists \tau.\, \Psi(\ell) = [\tau]$$

$$\Psi \vdash^{\ell} (\ell', \mathsf{some}(\tau, \tau')) \triangleq \Psi(\ell) = [\tau, \tau', \Psi(\ell')] \qquad h ::= \tau \mid \tau, \tau, h$$

$$\Psi \vdash \Sigma \triangleq \vdash \Sigma \wedge dom(\Sigma) = dom(\Psi) \wedge \forall \ell \in dom(\Psi).\, \Psi \vdash^{\ell} \Sigma(\ell) \qquad \Psi : \ell \to h$$

$$\Sigma : (k, \Psi) \triangleq \vdash \Psi \wedge \Psi \vdash \Sigma \wedge \forall \ell \in dom(\Sigma).\, (j, \Psi, \Sigma, \ell) \in \mathcal{VH}^{L}_{\mathsf{sim}}[\![\Psi(\ell)]\!]$$

Fig. 9. Vigilance for Simple Typing: Value-Log Type Satisfaction

For base types, ℓ belongs to the relation $\mathcal{V}^{L}_{\mathsf{sim}}[\![C]\!]$ if $\mathsf{pointsto}(\Sigma, \ell)$ is a value of the expected form. Since pairs are evaluated eagerly, they are never wrapped by extra types in the store, so the relation for pairs, $\mathcal{V}^{L}_{\mathsf{sim}}[\![\tau_1 \times \tau_2]\!]$ contains only labels that map to label pairs, and as usual, the components of the pair must belong to τ_1 and τ_2, respectively.

For function types, a function usually belongs to $\mathcal{V}^{L}_{\mathsf{sim}}[\![\tau_1 \to \tau_2]\!]$ if, when applied in some future world — when there are fewer steps left and the value log and type history potentially contain more labels — to a value that behaves like τ_1, it produces a result that behaves like τ_2. Our definition is slightly different: since we support subsumption and since applications in our language are annotated with type assertions, we consider applications in which the assertion τ_0 is any supertype of the result type τ_2, and require that the result behave like τ_0.[7]

For the dynamic type, $\mathcal{V}^{L}_{\mathsf{sim}}[\![*]\!]$ is an untagged union over base types $\mathcal{V}^{L}_{\mathsf{sim}}[\![C]\!]$, pairs of dynamic types $\mathcal{V}^{L}_{\mathsf{sim}}[\![* \times *]\!]$, and functions between dynamic types $\mathcal{V}^{L}_{\mathsf{sim}}[\![* \to *]\!]$. Since these types are not structurally smaller than $*$, step-indexing becomes crucial. For well-foundedness, an expression that behaves like $*$ for k steps is only required to behave like one of the types in the union for $k - 1$.

To extend this characterization to expressions, we define the expression relation $\mathcal{E}^{L}_{\mathsf{sim}}[\![\tau]\!]$. An expression e behaves like type τ if it does not terminate within the step-index budget, if it runs to an expected error, or if it produces a value that belongs to $\mathcal{V}^{L}_{\mathsf{sim}}[\![\tau]\!]$.

The logical relation defined thus far is mostly standard. We now consider how to ensure that every value also satisfies all the types from casts that were evaluated to produce that value. Note that all values that flow through casts are entered into the value log Σ. Thus Σ is analogous to a dynamically allocated (immutable) store and we can take inspiration from models of dynamically allocated (immutable) references[Ahmed 2004; Reddy and Yang 2003] to (1) keep track of the run-time typing histories of values in a type history Ψ, just as models of references keep track of the types of references in a store typing, and (2) ensure that values in Σ satisfy the run-time typing histories in Ψ, just as models of references ensure that the store S satisfies the store typing.

Thus, as the highlighted parts in Fig. 8 show, we set up a Kripke logical relation[Ahmed 2004; Pitts and Stark 1998] indexed by worlds comprised of a step-index k and a type history Ψ, which is a mapping from labels ℓ to run-time typing histories h that are essentially lists of types. We define a world accessibility relation $(j, \Psi') \sqsupseteq (k, \Psi)$, which says that (j, Ψ') is a future world accessible from (k, Ψ') if $j \leq k$ (we may have potentially fewer steps available in the future) and the future type history Ψ' may have more entries than Ψ. Whenever we consider future logs Σ', we require that there is a future world $(j, \Psi') \sqsupseteq (k, \Psi)$ such that the value log satisfies the typing history $\Sigma' : (j, \Psi')$. Where our relation differs from the standard treatment of state is in the constraints placed on Σ by Ψ via the value-log type-satisfaction relation $\Sigma : (k, \Psi)$, defined in Fig. 9.

[7]The logical relation is well founded despite the use of τ_0 here; it is defined by induction on the step index and (nested) induction on the structure of types, and in every instance of a use of subtyping, the step index is guaranteed to decrease.

In more detail, as Fig. 9 shows, our typing history Ψ associates with each label in the value log a run-time typing history h, where h is either a single type, indicating that the value was produced at that type, or h is two types τ, τ' appended onto another typing history h', indicating type obligations added by a cast expression that casts from τ' to τ. The head of a typing history h is the top-most type τ when h is τ, or τ when h is τ, τ', h'. We say that a value log Σ satisfies a world, written $\Sigma : (k, \Psi)$, when three things are true:

1. The type history must be syntactically well-formed: $\vdash \Psi$. The well-formedness constraint $\vdash \Psi$ ensures that each run-time typing history h is well formed. Because casts in our model may be coercive, they can only be expected to function appropriately when the value passed to the cast is of the appropriate (semantic) type. Since casts add types τ, τ' to the run-time typing history h of a value, this gives rise to a syntactic constraint that the "from" type τ' of such an entry matches (up to subsumption) the type that the casted value previously held in the history, namely $\mathrm{head}(h)$.

2. The value log must be well-formed given the type history: $\Psi \vdash \Sigma$. This requires certain syntactic constraints $\vdash \Sigma$ that are independent of Ψ, and that for each location ℓ, Σ should provide some value-log entry that is itself consistent with Ψ. The former constraint $\vdash \Sigma$ corresponds to the basic syntactic invariants preserved by the operational semantics: casted values are always compatible (due to the $v \propto_L^{cast} \tau$ checks performed by the cast evaluation rules) and are never pairs because our pairs are evaluated eagerly. For the latter, when the entry $\Psi(\ell)$ does not record a cast, $\Psi \vdash^{\ell} (v, \mathsf{none})$ specifies that the entry $\Psi(\ell)$ must be just τ, ie it does not include any type obligations added by a cast. If the entry $\Psi(l)$ does record a cast, then $\Psi \vdash^{\ell} (v, \mathsf{some}\tau, \tau')$ specified that the recorded types τ, τ' must match those in $\Psi(\ell)$, and the casted location ℓ' must itself be well formed with respect to the remaining entries in the run-time typing history $\Psi(\ell')$.

3. The values in the log must satisfy their run-time typing history. The core semantic condition of value-log type satisfaction is that $\Sigma(\ell)$ must behave like each type τ in its run-time typing history $\Psi(\ell)$. But we cannot simply ask that $\Sigma(\ell) \in \mathcal{V}^L_{\mathsf{sim}}[\![\tau]\!]$ for each $\tau \in \Psi(\ell)$. Since casts in our model may be coercive, they can only be expected to function appropriately when the value passed to the cast is of the appropriate (semantic) type. Because $\mathcal{V}^L_{\mathsf{sim}}[\![\tau_1 \to \tau_2]\!]$ quantifies over values $v \in \mathcal{V}^L[\![\tau_1]\!]$, if we were to take the above approach, a function cast from $\tau_1 \to \tau_2$ to τ' would need to behave well when applied to an argument $v \in \mathcal{V}^L_{\mathsf{sim}}[\![\tau_1]\!]$. Since a cast on a function must ensure that the function's actual argument v belongs to the type expected by the original function, it must semantically perform a cast equivalent to $\mathsf{cast}\,\{\tau_1 \Leftarrow \mathit{dom}(\tau')\}\,v$[8] to be well formed, which one would not expect to be true in general.

To properly incorporate this constraint, we define typing-history relations that specify when a value or expression behaves like multiple types at once[9]. These relations, $\mathcal{VH}^L_{\mathsf{sim}}[\![\overline{\tau}]\!]$ and $\mathcal{EH}^L_{\mathsf{sim}}[\![\overline{\tau}]\!]$ are given in the top of Fig. 10. For a nonempty list of types $\overline{\tau}$, the relation is defined inductively over the first type in the list, following a similar structure to $\mathcal{V}^L_{\mathsf{sim}}[\![\tau]\!]$. When τ is a base type C, the value typing-history relation $\mathcal{VH}^L_{\mathsf{sim}}[\![C, \overline{\tau}]\!]$ contains any ℓ such that $\mathsf{pointsto}(\Sigma, \ell)$ is in $\mathcal{V}^L_{\mathsf{sim}}[\![\tau]\!]$ for each $\tau \in [C, \overline{\tau}]$. As in the value relation, since casts on pairs are evaluated eagerly, $\mathcal{VH}^L_{\mathsf{sim}}[\![\tau_1 \times \tau_2, \overline{\tau}]\!]$ contains only pairs $\langle \ell_1, \ell_2 \rangle$ whose components inductively satisfy all the appropriate types.

As discussed above, $\mathcal{VH}^L_{\mathsf{sim}}[\![\tau_1 \to \tau_2, \overline{\tau}]\!]$ requires a function to behave well only when it is given an argument $v \in \mathcal{V}^L_{\mathsf{sim}}[\![\tau_1]\!]$. As in the $\mathcal{V}$ relation, it must also behave well when the application is annotated with any $\tau_0 \Rrightarrow \tau_2$. Behaving "well" means that an application, evaluated with a future

[8] In our reduction semantics, this constraint is ensured by an expression of the form $\mathsf{mon}\,\{\tau_1 \Leftarrow \mathit{dom}(\tau')\}\,v$ for the sake of Transient, which does not perform the expected checks here; see §5.

[9] An arbitrary list of types used as this index is more general than the grammar of h, but we will freely interconvert them, since the syntax of h is a subset of that of $\overline{\tau}$.

$$\mathcal{VH}^L_{\mathsf{sim}}[\![C, \tau_2, \ldots \tau_n]\!] \triangleq \{(k, \Psi, \Sigma, \ell) \mid \forall \tau \in [C, \tau_2, \ldots \tau_n].\ (k, \Psi, \Sigma, \ell) \in \mathcal{V}^L_{\mathsf{sim}}[\![\tau]\!]\}$$

$$\begin{aligned}\mathcal{VH}^L_{\mathsf{sim}}[\![\tau_1' \times \tau_1'', \tau_2, \ldots \tau_n]\!] \triangleq \{(k, \Psi, \Sigma, \ell) \mid \Sigma(\ell) = (\langle \ell_1, \ell_2 \rangle, \_)\\ \wedge\ (k, \Psi, \Sigma, \ell_1) \in \mathcal{VH}^L_{\mathsf{sim}}[\![\tau_1', fst(\tau_2), \ldots fst(\tau_n)]\!]\\ \wedge\ (k, \Psi, \Sigma, \ell_2) \in \mathcal{VH}^L_{\mathsf{sim}}[\![\tau_1'', snd(\tau_2), \ldots snd(\tau_n)]\!]\}\end{aligned}$$

$$\begin{aligned}\mathcal{VH}^L_{\mathsf{sim}}[\![\tau_1' \to \tau_1'', \tau_2, \ldots \tau_n]\!] \triangleq \{(k, \Psi, \Sigma, \ell) \mid \forall (j, \Psi') \sqsupseteq (k, \Psi),\ \Sigma' \supseteq \Sigma,\ \ell_v,\ \tau_0 \gtrdot \tau_2.\\ (j, \Psi', \Sigma', \ell_v) \in\ \mathcal{V}^L_{\mathsf{sim}}[\![\tau_1']\!] \wedge \Sigma' : (j, \Psi')\\ \Rightarrow (j, \Psi', \Sigma', \mathsf{app}\{\tau_0\}\ \ell\ \ell_v) \in \mathcal{EH}^L_{\mathsf{sim}}[\![[\tau_0, cod(\tau_2), \ldots cod(\tau_n)]]\!]\}\end{aligned}$$

$$\begin{aligned}\mathcal{VH}^L_{\mathsf{sim}}[\![*, \tau_2, \ldots \tau_n]\!] \triangleq \{(k, \Psi, \Sigma, \ell) \mid (k-1, \Psi, \Sigma, \ell) \in \mathcal{VH}^L_{\mathsf{sim}}[\![C, \tau_2, \ldots \tau_n]\!] \cup\\ \mathcal{VH}^L_{\mathsf{sim}}[\![* \times *, \tau_2, \ldots, \tau_n]\!] \cup \mathcal{VH}^L_{\mathsf{sim}}[\![* \to *, \tau_2, \ldots, \tau_n]\!]\}\end{aligned}$$

$$\begin{aligned}\mathcal{EH}^L_{\mathsf{sim}}[\![\overline{\tau}]\!] \triangleq \{(k, \Psi, \Sigma, e) \mid \forall j \le k.\ \forall \Sigma' \supseteq \Sigma, e'.\ (\Sigma, e) \longrightarrow^j_L (\Sigma', e') \wedge \mathsf{irred}(e')\\ \Rightarrow (e' = \mathsf{Err}^\bullet \vee (\exists (k-j, \Psi') \sqsupseteq (k, \Psi).\ \Sigma' : (k-j, \Psi') \wedge (k-j, \Psi', \Sigma', e') \in \mathcal{VH}^L_{\mathsf{sim}}[\![\overline{\tau}]\!]))\}\end{aligned}$$

$$\mathcal{G}^L_{\mathsf{sim}}[\![\Gamma]\!] \triangleq \{(k, \Psi, \Sigma, \gamma) \mid dom(\Gamma) = dom(\gamma) \wedge \forall x.(k, \Psi, \Sigma, \gamma(x)) \in \mathcal{V}^L_{\mathsf{sim}}[\![\Gamma(x)]\!]\}$$

$$[\![\Gamma \vdash_{\mathsf{sim}} e : \tau]\!]^L \triangleq\ \forall (k, \Psi, \Sigma, \gamma) \in \mathcal{G}^L_{\mathsf{sim}}[\![\Gamma]\!].\ \Sigma : (k, \Psi)\ \Rightarrow (k, \Psi, \Sigma, \gamma(e)) \in \mathcal{E}^L_{\mathsf{sim}}[\![\tau]\!]$$

Fig. 10. Vigilance for Simple Typing: Typing History and Top-Level Relations

store $\Sigma' : (j, \Psi') \sqsupseteq (k, \Psi)$ should behave like all the types $\tau_0, cod(\overline{\tau})$. Since this is an expression, we define the $\mathcal{EH}^L_{\mathsf{sim}}[\![\overline{\tau}]\!]$ relation to characterize expressions as behaving like several types at once; since this only matters when the expression reduces to a value, it is precisely the same as the $\mathcal{E}$ relation, except that it is indexed by $\overline{\tau}$ rather than τ, and it requires the eventual value to be in $\mathcal{VH}^L_{\mathsf{sim}}[\![\overline{\tau}]\!]$ rather than $\mathcal{V}^L_{\mathsf{sim}}[\![\tau]\!]$. Finally, as in the value relation, $\mathcal{VH}^L_{\mathsf{sim}}[\![*, \overline{\tau}]\!]$ is an untagged union over base types $\mathcal{VH}^L_{\mathsf{sim}}[\![C, \overline{\tau}]\!]$, pairs of dynamic types $\mathcal{VH}^L_{\mathsf{sim}}[\![* \times *, \overline{\tau}]\!]$, and functions between dynamic types $\mathcal{VH}^L_{\mathsf{sim}}[\![* \to *, \overline{\tau}]\!]$, at step index $k-1$.

In typical fashion, we generalize to open expressions at the bottom of Fig. 10. In detail, $[\![\Gamma \vdash_{\mathsf{sim}} e : \tau]\!]^L$ says that an expression e that type checks in context Γ behaves like τ under L when, given a substitution γ that behaves like Γ, $\gamma(e)$ behaves like τ. Moreover, a substitution γ, mapping free variables x to labels ℓ in Σ, behaves like Γ when for each $x : \tau$ in Γ, $\gamma(x)$ behaves like $\Gamma(x)$.

Note. Abstracting the construction of the vigilance relation for simple typing and returning to the recipe from §2.5, the starting point for constructing a vigilance relation is a standard type soundness model. That basic model should be extended with a world structure that captures typing histories, such as Ψ. In many cases, we conjecture it is possible to cartesian product the additional world structure onto the world structure used for soundness. The next step is the incorporation of the additional world structure into the $\mathcal{V}$ and $\mathcal{E}$ relations. The resulting relations, $\mathcal{VH}$ and $\mathcal{EH}$, should be liftings of $\mathcal{V}$ and $\mathcal{E}$ but their definitions require some "semantic thinking" for managing the extended world (just as the definitions of $\mathcal{V}$ and $\mathcal{E}$ do). Finally, store satisfaction, $\Sigma : (k, \Psi)$, should be strengthened with extra conditions as listed above, reflecting the extension of the world structure. Importantly, store satisfaction should entail that the semantics enforce typing histories.

With our vigilance logical relation in place, we formally establish that the combination of Uni and Natural semantics are vigilant for simple typing iff all well typed GTL expressions t translate to ICTL expressions e in the vigilance relation.

Theorem 4.1 (Uni-Natural Is Vigilant for Simple Typing). *If* $\Gamma \vdash_{\mathsf{Uni}} t : \tau \rightsquigarrow e$ *then* $[\![\Gamma \vdash_{\mathsf{sim}} e : \tau]\!]^{\mathsf{Natural}}$.

$$
\begin{array}{rcl}
K & ::= & \mathsf{Nat} \mid \mathsf{Int} \mid \mathsf{Bool} \mid * \times * \mid * \to * \mid * \\
\Gamma & ::= & \cdot \mid \Gamma, (x{:}K) \\
e & ::= & x \mid n \mid i \mid \mathsf{True} \mid \mathsf{False} \mid \lambda(x{:}\tau).\, e \mid \langle e, e \rangle \mid \mathsf{app}\{\tau\}\, e\, e \\
 & \mid & \mathsf{fst}\{\tau\}\, e \mid \mathsf{snd}\{\tau\}\, e \mid \mathit{binop}\, e\, e \\
 & \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{cast}\,\{\tau \Leftarrow \tau\}\, e
\end{array}
$$

$\boxed{\Gamma \vdash_{\mathsf{tag}} e : K}$ (selected rules)

$$
\textsc{T-App}\ \frac{\Gamma_0 \vdash_{\mathsf{tag}} e_0 : * \to * \qquad \Gamma_0 \vdash_{\mathsf{tag}} e_1 : K}{\Gamma_0 \vdash_{\mathsf{tag}} \mathsf{app}\{\tau\}\, e_0\, e_1 : \lfloor \tau \rfloor}
$$

Fig. 11. Tag Typing for λ^{ICTL}

The theorem is a direct consequence of whether the Natural semantics satisfies the fundamental property of the vigilance relation for simple typing. As described in §3, given a type-preserving translation, such as Uni, the question of whether the combination of the translation and a semantics is vigilant for a type system reduces to whether the semantics enforces the typing histories. The fundamental property entails exactly that. First, values satisfy the types ascribed to them by the simple type system; well-typed terms $\vdash_{\mathsf{sim}} e : \tau$ are in $\mathcal{E}^L_{\mathsf{sim}}[\![\tau]\!]$, which says that if e runs to a value, then the value must behave like τ. Second, the fundamental property entails that values satisfy the types of all casts that were evaluated to produce them; if e runs to a value with value log Σ', then there is a future world $(k - j, \Psi') \sqsupseteq (k, \Psi)$ such that $\Sigma' : (k - j, \Psi')$.

Theorem 4.2 (Natural Satisfies the Fundamental Property for Simple Typing). *If* $\Gamma \vdash_{\mathsf{sim}} e : \tau$ *then* $[\![\Gamma \vdash_{\mathsf{sim}} e : \tau]\!]^{\mathsf{Natural}}$.

As for the combination of Uni and Transient, we can show that they are not vigilant for simple typing with the counter-example from §2.

Theorem 4.3 (Uni-Transient Is Not Vigilant for Simple Typing). *There are* Γ, t, K, e *such that* $\Gamma \vdash_{\mathsf{Uni}} t : \tau \rightsquigarrow e$ *and* $\neg[\![\Gamma \vdash_{\mathsf{sim}} e : \tau]\!]^{\mathsf{Transient}}$.

5 TRANSIENT IS MORE THAN TAG CHECKING

Transient does not use proxies and, as a result, it does not enforce all the types from casts it encounters during the evaluation of an ICTL program. For instance, consider $p = \langle -1, -1 \rangle$, $f = \lambda(x{:}\mathsf{Int} \times \mathsf{Int}) \to (\mathsf{Nat} \times \mathsf{Nat}).\, x$, and $t = f\ p$. In words, consider a function that takes a pair of Ints and returns it as a pair of Nats, and apply the function to the pair $\langle -1, -1 \rangle$. Under Uni, t becomes:

$$e = \mathsf{app}\{\mathsf{Nat} \times \mathsf{Nat}\}\ (\lambda(x{:}\mathsf{Int} \times \mathsf{Int}).\, \mathsf{cast}\,\{\mathsf{Nat} \times \mathsf{Nat} \Leftarrow \mathsf{Int} \times \mathsf{Int}\}\ x)\ \langle -1, -1 \rangle$$

Interestingly, since Transient only checks the top-level constructor of types on casts, $e \hookrightarrow_{\mathsf{T}} \langle -1, -1 \rangle$. However, the final result of e is also the result of f which is supposed to be a $\mathsf{Nat} \times \mathsf{Nat}$ according to f's casts. Hence, Transient is not vigilant for simple typing:

Theorem 5.1 (Uni-Transient Is Not Vigilant for Simple Typing). *There are* Γ, t, e, τ *such that* $\Gamma \vdash_{\mathsf{Uni}} t : \tau \rightsquigarrow e$ *and* $\neg[\![\Gamma \vdash_{\mathsf{sim}} e : \tau]\!]^{\mathsf{Transient}}$.

However, as previous work hints [Greenman and Felleisen 2018; Greenman et al. 2019a; Vitousek et al. 2017], Transient should enforce the tag typing history of values produced during the evaluation of an ICTL expression. Following the recipe from §2.5, we formalize this relation between Transient and the enforcement of tags as a vigilance property by: (i) adjusting the type system of λ^{GTL} and λ^{ICTL} to tag typing; (ii) reusing Uni to map λ^{GTL} expressions to λ^{ICTL} expressions with the same tags — since Uni preserves simple typing, it also preserves tag typing; and (iii) modifying the vigilance logical relation from §4 to capture the semantics of ICTL tags rather than ICTL types.

Fig. 11 sketches how we adjust the simple type system of λ^{ICTL} to tag typing . The tag typing rules, which relate an expression e with its top-level constructor K, are entirely unsurprising. The most interesting one is that for applications allows arguments at any type, since the type of a

$$\mathcal{V}^L_{\mathsf{tag}}[\![* \to *]\!] \triangleq \{(k, \Psi, \Sigma, \ell) \mid \forall (j, \Psi') \sqsupseteq (k, \Psi),\ \Sigma' \supseteq \Sigma,\ \ell_v,\ \tau_0.\ \Sigma' : (j, \Psi') \wedge (j, \Psi', \Sigma', \ell_v) \in \mathcal{V}^L_{\mathsf{tag}}[\![*]\!].\ \Rightarrow (j, \Psi', \Sigma', \mathsf{app}\{\tau_0\}\ \ell\ \ell_v) \in \mathcal{E}^L_{\mathsf{tag}}[\![\lfloor \tau_0 \rfloor]\!]\}$$

Fig. 12. Function Case of the Value Vigilance Relation for Tag Typing

function is simply the procedure tag that does not contain any information about the function's domain. As expected, every expression that simply type checks also tag type checks:

Theorem 5.2 (Simple Typing Implies Tag Typing). *If* $\Gamma \vdash_{\mathsf{sim}} e : \tau$, *then* $\lfloor \Gamma \rfloor \vdash_{\mathsf{tag}} e : \lfloor \tau \rfloor$.

where $\lfloor \cdot \rfloor$ returns the tag of a given type τ, and which we lift in a straightforward fashion to type environments Γ. Finally, the Uni translations is tag-preserving, meaning it translates a λ^{GTL} expression t with type τ to a λ^{ICTL} expression e with type $\lfloor \tau \rfloor$:

Theorem 5.3 (Uni is Tag-Preserving). *If* $\Gamma \vdash_{\mathsf{Uni}} t : \tau \rightsquigarrow e$ *then* $\lfloor \Gamma \rfloor \vdash_{\mathsf{tag}} e : \lfloor \tau \rfloor$.

A vigilance relation for tag typing requires modifying the vigilance relation for simple typing from §4 to obtain a relational interpretation of tag types. The modification is rather straightforward and reflects the differences between the simple and tag type systems for λ^{ICTL}. The full details are in the supplemental material; here we highlight a few key points. In general, the value and expression relations for tag typing, i.e., $\mathcal{V}^L_{\mathsf{tag}}$, $\mathcal{VH}^L_{\mathsf{tag}}$, $\mathcal{E}^L_{\mathsf{tag}}$, $\mathcal{EH}^L_{\mathsf{tag}}$, are identical to the corresponding relations for simple typing except that types are restricted to tags. For example, the cases for Nat, Int, Bool, $* \times *$, and $*$ in $\mathcal{V}^L_{\mathsf{tag}}$ and $\mathcal{VH}^L_{\mathsf{tag}}$ follow the exact template of the corresponding cases in $\mathcal{V}^L_{\mathsf{sim}}$ and $\mathcal{VH}^L_{\mathsf{sim}}$ modulo the use of $\lfloor \cdot \rfloor$ as needed to turn types into tags. However, the cases for the procedure tag require additional care. Fig. 12 shows the cases of $\mathcal{V}^L_{\mathsf{tag}}$ for $* \to *$. In addition to the restriction of types to tags, the type assertion for the application is about an arbitrary type τ_0, not a supertype of the codomain of the function as in $\mathcal{V}^L_{\mathsf{sim}}$. Additionally, the relation does not require that the application behaves according to the codomain $*$, but instead the tag $\lfloor \tau_0 \rfloor$ of the annotation. These extra modifications reflect that the relation aims to capture the meaning of tags not types. The $\mathcal{VH}^L_{\mathsf{tag}}[\![* \to *, \ldots K_n]\!]$ case is analogous.

With tag typing and the vigilance relation in hand, we can show that the combination of the Uni translation and Transient is indeed vigilant for tag typing:

Theorem 5.4 (Uni-Transient Is Vigilant for Tag Typing).
If $\Gamma \vdash_{\mathsf{Uni}} t : \tau \rightsquigarrow e$ *then* $[\![\Gamma \vdash_{\mathsf{tag}} e : \lfloor \tau \rfloor]\!]^{\mathsf{Transient}}$.

As for theorem 4.1, theorem 5.4 derives from tag-preservion for Uni and the fundamental property of the vigilance relation for tag typing.

Theorem 5.5 (Transient Satisfies the Fundamental Property of the Vigilance Relation for Tag Typing). *If* $\Gamma \vdash_{\mathsf{tag}} e : K$ *then* $[\![\Gamma \vdash_{\mathsf{tag}} e : K]\!]^{\mathsf{Transient}}$.

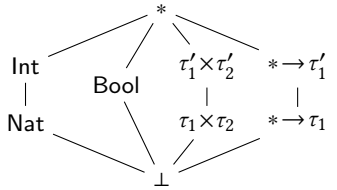
Despite Theorem 5.2, tag types are not semantically "weaker" than simple types: a function with type $* \to *$ can be used safely in more contexts than a function with type $\tau \to *$. Because of this difference tag typing is unsound for Natural. For instance, the GTL expression $\lambda(x : * \times *) \to *.\ \mathsf{fst}\ x$ translates under Uni to $e = \lambda(x : * \times *).\ \mathsf{fst}\{*\}\ x$ which has tag $* \to *$. The interpretation of $* \to *$ in the vigilance relation for tag typing requires that e applied to *any* e' in the interpretation of $*$ is well behaved. When e is applied to any e' that is not a pair, under Natural this application will throw a soundness error Wrong. In contrast, the same expression produces a type error TypeErr in Transient since Transient uses a type assertion to check the tag of function arguments. At core, Natural relies on stronger type invariants to ensure safety, while Transient uses dynamic checks to ensure safety in more contexts. Consequently, Natural is not vigilant for tag typing:

$\boxed{\Gamma \vdash_{\mathsf{tru}} e : \tau}$ (selected rules)

$$\frac{\Gamma,\ (x{:}K) \vdash_{\mathsf{tru}} e : \tau}{\Gamma \vdash_{\mathsf{tru}} \lambda(x{:}K).\, e : * \to \tau} \qquad \frac{\Gamma \vdash_{\mathsf{tru}} e_1 : * \to \tau_1 \quad \Gamma \vdash_{\mathsf{tru}} e_2 : \tau_2}{\Gamma \vdash_{\mathsf{tru}} \mathsf{app}\{K\}\, e_1\, e_2 : K \sqcap \tau_1}$$

$$\frac{\Gamma \vdash_{\mathsf{tru}} e_1 : \bot \quad \Gamma \vdash_{\mathsf{tru}} e_2 : \tau_2}{\Gamma \vdash_{\mathsf{tru}} \mathsf{app}\{K\}\, e_1\, e_2 : \bot} \qquad \frac{\Gamma \vdash_{\mathsf{tru}} e_b : \mathsf{Bool} \quad \Gamma \vdash_{\mathsf{tru}} e_1 : \tau_1 \quad \Gamma \vdash_{\mathsf{tru}} e_2 : \tau_2}{\Gamma \vdash_{\mathsf{tru}} \mathsf{if}\ e_b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau_1 \sqcup \tau_2}$$

$\tau ::= \mathsf{Nat} \mid \mathsf{Int} \mid \mathsf{Bool} \mid \tau \times \tau \mid * \to \tau \mid * \mid \bot$



$$\frac{\Gamma \vdash_{\mathsf{tru}} e : \tau}{\Gamma \vdash_{\mathsf{tru}} \mathsf{cast}\,\{K_2 \Leftarrow K_1\}\ e_0 : K_2 \sqcap K_1 \sqcap \tau}$$

Fig. 13. Truer Typing for λ^{ICTL}

Theorem 5.6 (Uni-Natural Is Not Vigilant for Tag Typing). *There are* Γ, t, K, e *such that* $\Gamma \vdash_{\mathsf{Uni}} t : \tau \rightsquigarrow e$ *and* $\neg[\![\lfloor \Gamma \rfloor \vdash_{\mathsf{tag}} e : \lfloor \tau \rfloor]\!]^{\mathsf{Natural}}$.

5.1 A Truer Type System for λ^{ICTL}

While each individual Transient cast checks only a tag, because the Uni-Transient combination is vigilant for tag typing extra information about a value is available. For example, consider the function $f = \lambda(x{:}*\times*).\, x$. Under the tag type system, f type checks at $* \to *$. From this type, we can deduce only that f is well-behaved when given any argument, and that it makes no promises about its result. However, vigilance for tag typing implies that Transient also checks that the argument of f is a pair. Consequently, we should be able to conclude that the return type of f is really $*\times*$. As this example hints, there should an alternative type system that the Transient approach can actually enforce. That is, we can capture *more precise* static information than tags with *no change* to the dynamics. As an exercise, we make this extra static information manifest in a *truer* type system for λ^{ICTL}. In the remainder of this section, we describe this type system, use vigilance to examine it, and demonstrate how it enables the elision of some tag checks that Transient performs.

Fig. 13 presents the truer type system. A key distinction with the , its rules assume a restricted λ^{ICTL} syntax where type ascriptions are tag K. Similarly, type environments Γ map variables to tags. However, truer typing deduces more precise types τ than tags. These differ from simple types in two major ways. First, the domain of function types is always $*$. After all, in the Transient approach, the type assertions injected in the body of a function – including the tag check of its argument — guarantee that the function can handle any argument. Second, truer typing can deduce that some expressions raise a run-time type error due to incompatible tag checks, and hence, truer types include $\bot$. This inclusion of $\bot$ allows us to define a full subtyping lattice $\leq$ on truer transient types, as shown in the upper right portion of Fig. 13. Like other systems with subsumption rules for subtyping, the truer type system includes a subsumption rule, but for the subtyping lattice $\leq$.

The typing rule for anonymous functions type checks the body of a function under the assumption that the function's argument has the ascribed tag. But, as discussed above, the domain of the function is $*$ because applications implicitly check the argument of a lambda against this tag annotation. Dually, the rule for applications admits function arguments that typecheck at any tag.

Because applications perform a tag check on the result of the application, rather than typing the entire expression at the codomain of the function τ_1, the truer type system seeks to take advantage of the fact that the result of the application satisfies *both* τ_1 *and* K. For that, the typing rule calculates the result type as the greatest lower bound $K \sqcap \tau_1$. If τ_1 is $\bot$, a special application rule propagates it to the result type of the application. Similar to the non-$\bot$ application rule, the rule for cast

$\boxed{\Gamma \vdash_{\mathsf{Flow}} t \Rightarrow \tau \rightsquigarrow e}$ (selected rules)

$$\frac{\Gamma,\ (x{:}K) \vdash_{\mathsf{Flow}} t \Leftarrow^{+} \tau \rightsquigarrow e : \tau'}{\Gamma \vdash_{\mathsf{Flow}} \lambda(x{:}K) \to \tau.\, t \Rightarrow * \to \tau \rightsquigarrow \lambda(x{:}K).\, e : * \to \tau'} \qquad \frac{\Gamma \vdash_{\mathsf{Flow}} t \Leftarrow \tau \rightsquigarrow e : \tau'}{\Gamma \vdash_{\mathsf{Flow}} t \Leftarrow^{+} \tau \rightsquigarrow e : \tau'} \qquad \frac{\neg(\exists e, \tau'.\ \Gamma \vdash_{\mathsf{Flow}} t \Leftarrow \tau \rightsquigarrow e : \tau') \quad \Gamma \vdash_{\mathsf{Flow}} t \Leftarrow^{\Rightarrow} \tau \rightsquigarrow e : \tau'}{\Gamma \vdash_{\mathsf{Flow}} t \Leftarrow^{+} \tau \rightsquigarrow e : \tau'}$$

$$\frac{\Gamma \vdash_{\mathsf{Flow}} t \Leftarrow^{+} (\tau \setminus \lfloor \tau \rfloor) \times * \rightsquigarrow e : \tau_1 \times \tau_2}{\Gamma \vdash_{\mathsf{Flow}} \mathsf{fst}\, t \Leftarrow \tau \rightsquigarrow \mathsf{fst}\{\lfloor \tau \rfloor\}\, e : \tau_1 \sqcap \lfloor \tau \rfloor} \qquad \frac{\Gamma \vdash_{\mathsf{Flow}} t \Rightarrow \tau' \rightsquigarrow e : \tau'' \quad \tau' \leq \tau}{\Gamma \vdash_{\mathsf{Flow}} t \Leftarrow^{\Rightarrow} \tau \rightsquigarrow e : \tau''} \qquad \frac{\Gamma \vdash_{\mathsf{Flow}} t \Rightarrow \tau' \rightsquigarrow e : \tau'' \quad \tau' \not\leq K}{\Gamma \vdash_{\mathsf{Flow}} t \Leftarrow^{\Rightarrow} K \rightsquigarrow \mathsf{cast}\,\{K \Leftarrow \lfloor \tau' \rfloor\}\, e : K \sqcap \lfloor \tau' \rfloor \sqcap \tau''}$$

Fig. 14. Flow Sensitive Translation from λ^{GTL} to λ^{ICTL}

expressions refines the type of its result type with the tag check from the cast. Finally, conditionals typecheck at the least upper bound, $\tau_1 \sqcup \tau_2$, of *both* branches.

Because of its flow-sensitive nature, the truer type system accepts just as many programs as the tag type system, but calculates more precise types for them:

Theorem 5.7 (Tag Typing Implies Truer Typing). *Suppose that* $\Gamma \vdash_{\mathsf{tag}} e : K$. *Then there exists some* $\tau \leq K$ *such that* $\Gamma \vdash_{\mathsf{tru}} e : \tau$.

Similarly, since simple typing implies tag typing, it also implies truer typing:

Corollary 5.8 (Simple Typing Implies Truer Transient Typing). *If* $\Gamma \vdash_{\mathsf{sim}} e : \tau$, *then* $\lfloor \Gamma \rfloor \vdash_{\mathsf{tru}} \lfloor e \rfloor : \tau'$ *where* $\tau' \leq \lfloor \tau \rfloor$.

5.2 Translating λ^{GTL} to λ^{ICTL} for Truer

Using vigilance for examining truer typing is not as straightforward as for tag typing. Specifically, in the case of tag typing, we can use the Uni translation as is. However, in order to get more precise types while maintaining an algorithmic presentation of λ^{GTL} with truer typing, we need a new translation. In response, we design a bidirectional type system and translation from λ^{GTL} to λ^{ICTL}, which allows us to capitalize on truer typing's ability to take advantage of the tag checks from casts to refine the types of expressions.

Fig. 14 presents a few salient rules of the judgments that define the type checker for λ^{GTL} expressions and their *flow sensitive* translation to λ^{ICTL}. Unlike the Uni translation, Flow produces a type τ' for the translated term e. The translation is defined so that $\tau' \leq \tau$ in the subtyping lattice, which implies that the evaluation of e with the Transient semantics must at least enforce the types of t (by the subsumption rule), but may also enforce a "stronger" type τ'. Both τ and τ' are a safe basis for reasoning about the behavior of a well-typed expression (an example is given in §5.4).

The "infers" judgment of the bidirectional translation ($\Gamma \vdash_{\mathsf{Flow}} t \Rightarrow \tau \rightsquigarrow e' : \tau'$) is similar to that of the Uni translation from §3; given a type environment and a λ^{GTL} expression, it type checks t at type τ, translates it to the λ^{ICTL} expression e, and calculates the λ^{ICTL} type τ'. However, when t contains a type annotation—such as in the return type annotation of a function—rather than inserting a cast expression that is supposed to enforce the annotation, the judgment appeals to the "checks" judgment $\Gamma \vdash_{\mathsf{Flow}} t \Leftarrow^{+} \tau \rightsquigarrow e : \tau'$ that attempts to construct a translated function body e' that has type τ, and calculate its type τ'. After all, the truer type system is supposed to reflect the types that Transient enforces, and Transient only uses casts that perform tag checks.

The "checks" judgment itself employs two other judgments. $\Gamma \vdash_{\mathsf{Flow}} t \Leftarrow \tau \rightsquigarrow e : \tau'$ inserts an appropriate type ascription to an application or a pair projection, and delegates back to the "checks" judgment to construct a term that has the portion of τ that the ascription does not cover (see Fig. 17,

$$\mathcal{V}^L_{\mathsf{tru}}[\![* \to \tau]\!] \triangleq \{(k, \Psi, \Sigma, \ell) \mid \forall (j, \Psi').\ (k, \Psi) \sqsupseteq (k, \Psi),\ \Sigma' \supseteq \Sigma,\ \ell_v,\ \tau_0.$$
$$\Sigma' : (j, \Psi') \wedge (j, \Psi', \Sigma', \ell_v) \in \mathcal{V}^L_{\mathsf{tru}}[\![*]\!].\ \Rightarrow (j, \Psi', \Sigma', \mathsf{app}\{\tau_0\}\ \ell\ \ell_v) \in \mathcal{E}^L_{\mathsf{tru}}[\![\tau \sqcap \lfloor \tau_0 \rfloor]\!]\}$$

Fig. 15. Function Case of the Value Vigilance Relation for Truer Typing

bottom right, for the definition of $\cdot \setminus \cdot$). $\Gamma \vdash_{\mathsf{Flow}} t \Leftarrow^{\Rightarrow} \tau \rightsquigarrow e : \tau'$ applies to any expression where the previous judgment does not. It uses the "infers" judgment recursively to infer a type τ' for t that is either a subtype of τ, or a type whose tag it can cast to τ (if τ is a tag).

As an example of this translation, consider the λ^{GTL} expression

$$t = \lambda(x : * \times *) \to \mathsf{Nat} \times \mathsf{Nat}.\ \langle \mathsf{fst}\, x, \mathsf{snd}\, x \rangle$$

The "infers" judgments translates the body of the function , but doesn't simply try to insert a single cast to enforce Nat×Nat, like the HO translation judgment from §3 would. Instead, it delegates to the "checks" judgment which attempts to find a way to insert type assertions into the body of the function in order to construct a body that has the desired result type. Hence, with the help of the $\Leftarrow$ judgment, it asserts that the pair projections in the body of the function are naturals, per the tag of the required type, leading to the following translated λ^{ICTL} expression:

$$e = \lambda x : * \times *.\ \langle \mathsf{fst}\{\mathsf{Nat}\}\, x, \mathsf{snd}\{\mathsf{Nat}\}\, x \rangle$$

Importantly, just like the Uni translation, the Flow translation is type-preserving:

Theorem 5.9 (Flow Preserves Truer Types). *If* $\Gamma \vdash_{\mathsf{Flow}} t \Rightarrow \tau \rightsquigarrow e : \tau'$ *then* $\Gamma \vdash_{\mathsf{tru}} e : \tau$.

5.3 Flow and Transient are Vigilant for Truer Typing

The specifics of the truer type system require a few modifications to the vigilance logical relation from §4. Most of them are analogous to those for tag typing described above — the supplemental material includes all the details. The key difference between $\mathcal{V}^L_{\mathsf{tru}}$ and $\mathcal{VH}^L_{\mathsf{tru}}$ and $\mathcal{V}^L_{\mathsf{tag}}$ and $\mathcal{VH}^L_{\mathsf{tag}}$ is that the first are not restricted to tags but to truer types. To make the discussion concrete, Fig. 15 shows the case of $\mathcal{V}^L_{\mathsf{tru}}$ for truer function types, $\mathcal{V}^L_{\mathsf{tru}}[\![* \to \tau]\!]$. Similar to the corresponding case for tag typing in Fig. 12, there is no restriction on the annotation of the application. However, the result of the application is required to behave according not just to $\lfloor \tau_0 \rfloor$, but to $\tau \sqcap \lfloor \tau_0 \rfloor$, reflecting the truer typing for applications in Fig. 13. Finally, $\mathcal{V}^L_{\mathsf{tru}}$ comes with an extra case for $\bot$ given its important role in the truer type system.

It is worth noting that the vigilance relation for truer typing is stronger than the corresponding logical relation for type soundness. While the truer type system aims to reflect statically as much information as possible from the tag checks performed during the evaluation of a λ^{ICTL} expression, the static approximation in the conditional rule shows, as all type systems, it fails to do so accurately. As a result, if truer typing deduces that the result of a conditional has type $*$, then a semantics can ignore some of the checks the branches of the conditional are supposed to perform and still truer typing would be sound. However, vigilance requires that the semantics performs all the checks from the evaluated branch nevertheless.

With the Flow translation and the vigilance relation for true typing in hand, we can show that the combination of Flow and Transient is vigilant for truer typing:

Theorem 5.10 (Flow-Transient Is Vigilant for Truer Typing). *If* $\Gamma \vdash_{\mathsf{Flow}} t \Rightarrow \tau \rightsquigarrow e$ *then* $[\![\Gamma \vdash_{\mathsf{tru}} e : \tau]\!]^{\mathsf{Transient}}$.

The theorem follows from Flow being type preserving and the fundamental property for the vigilance relation for truer typing:

Theorem 5.11 (Transient Satisfies the Fundamental Property of the Vigilance Relation for Truer Typing). *If* $\Gamma \vdash_{\mathsf{tru}} e : \tau$ *then* $[\![\Gamma \vdash_{\mathsf{tru}} e : \tau]\!]^{\mathsf{Transient}}$.

$\boxed{\Gamma \vdash_{\mathsf{opt}} e : \tau \rightsquigarrow e}$ (selected rules)

$$\frac{\Gamma_0 \vdash_{\mathsf{opt}} e_0 : * \to \tau_1 \rightsquigarrow e_0' \quad \Gamma_0 \vdash_{\mathsf{opt}} e_1 : \tau_0' \rightsquigarrow e_1'}{\Gamma_0 \vdash_{\mathsf{opt}} \mathsf{app}\{K_1\}\, e_0\, e_1 : K_1 \sqcap \tau_1 \rightsquigarrow \mathsf{app}\{K_1 \setminus \tau_1\}\, e_0'\, e_1'} \qquad \frac{\Gamma_0 \vdash_{\mathsf{opt}} e_0 : \bot \rightsquigarrow e_0' \quad \Gamma_0 \vdash_{\mathsf{opt}} e_1 : \tau_0' \rightsquigarrow e_1'}{\Gamma_0 \vdash_{\mathsf{opt}} \mathsf{app}\{K_1\}\, e_0\, e_1 : \bot \rightsquigarrow \mathsf{app}\{K_1 \setminus \bot\}\, e_0'\, e_1'}$$

$$\frac{\Gamma_0 \vdash_{\mathsf{opt}} e_0 : \tau_0 \rightsquigarrow e_0'}{\Gamma_0 \vdash_{\mathsf{opt}} \mathsf{cast}\,\{K_1 \Leftarrow K_0\}\, e_0 : K_1 \sqcap K_0 \sqcap \tau_0 \rightsquigarrow \mathsf{cast}\,\{K_1 \setminus (K_0 \sqcap \tau_0) \Leftarrow K_0 \setminus \tau_0\}\, e_0'} \qquad K \setminus \tau = \begin{cases} * & \text{if } \tau \leq K \\ K & \text{otherwise} \end{cases}$$

Fig. 17. Truer Typing: Check-elision optimization for λ^{ICTL}

5.4 When are Transient Checks Truly Needed?

Since in Transient *all* elimination forms perform tag checks, even those in code with precise types, some of these checks are redundant. Vitousek et al. [2019] use a sophisticated whole-program constraint system to infer when Transient's tag checks may be elided due to static information that the type system computes. The static information the truer type system provides may very naturally be used to implement and prove correct a similar elision pass for Transient tag checks. Indeed, given truer, local type-based reasoning is sufficient for the elision of the same checks as Vitousek et al. [2019] except those on function arguments, which require a whole-program analysis.

For example, consider the variant of the example from §1 in Fig. 16. Here, the snippet defines two different type adapters for the function `segment`, with different truer types. Since calls to `png_crop` ensure a tag on each result, `segment_png_small` will produce PNGs, while `segment_png` will not. At each projection in `segment_png` and `segment_png_small`, Transient checks that the result is a PNG. This tag check is however only necessary in the case of `segment_png`, where it is not statically known that (due to other checks) a PNG would be produced. Precisely this difference between `segment_png` and `segment_png_small`, which allows eliding a check in one case but not the other, is reflected in their truer types!

```
let segment_png_small =
  λ (img:PNG) → PNG × PNG.
    let (a, b) = segment img
    png_crop a, png_crop b
let segment_png =
  λ (img:PNG) → * × *. segment img
let png1 : PNG = fst (segment_png (...))
let png2 : PNG = fst (segment_png_small (...))
```

Fig. 16. Two Type Adapters for an Image Library

In terms of the rules of the truer type system from Fig. 13, all rules that involve an expression that performs a check of a tag K strengthen the type of the expression to $K \sqcap \tau$. Hence, the tag check improves what can be statically known about the behavior of the expression in hand — rather than only knowing that it behaves according to τ, we also know that it behaves according to K. As a result, such a tag check is useful only when the strengthened type ($K \sqcap \tau \neq \tau$) is more precise than τ — that is, when it is not already known that the value in question would behave like a K ($\tau \not\leq K$).

Fig. 17 provides an overview of an elision pass for redundant tag checks. The judgment $\Gamma \vdash_{\mathsf{opt}} e : \tau \rightsquigarrow e$ consumes a typing environment and a λ^{ICTL} expression e, type checks e at τ the same way as the truer type system for λ^{ICTL}, and uses the deduced types to translate e to an equivalent expression e' without some redundant tag checks. In essence, the translation replaces a type ascription τ with $K \setminus \tau$ where K is a tag that the translated expression checks. In general, $K \setminus \tau$ denotes the tag check that is necessary to enforce K given that τ is already known — in the typing lattice, this is $*$ if $\tau \leqslant: K$, and K otherwise. The elision pass preserves contextual equivalence:

Theorem 5.12 (Check-elision soundness). *If* $\Gamma \vdash_{\mathsf{opt}} e : \tau \rightsquigarrow e'$, *then* $\Gamma \vdash_{\mathsf{tru}} e \approx^{\mathrm{ctx}} e' : \tau$.

6 RELATED WORK AND CONCLUSION

Other Systems. We conjecture that vigilance applies to gradually typed languages with features other than the ones herein with the help of standard logical relations techniques. For instance, logical relations for higher-order mutable state [Ahmed 2004] are thoroughly explored in the literature, and can be the foundation for applying vigilance to gradually typed languages with references. More generally, vigilance determines whether a typing history is enforced, not by prescribing how checks happen, but rather by requiring that the checks that do happen entail the semantics of the term's type. As a result, given a logical relation that describes the semantics of the types of a language, vigilance can be applied regardless of the mechanism of dynamic enforcement.

A number of gradual typing approaches aim to improve the performance of Natural gradual typing by *eliminating* unnecessary proxies. Space-efficient gradual typing [Herman et al. 2010] employs a coercion-based evaluation language, which plays the role of our ICTL, where coercions summarize stacks of type casts around a value. Threesomes [Siek and Wadler 2010] use a cast-based evaluation language where a cast contains a triplet of types instead of a pair. Such casts can be combined to reduce the number of dynamic checks. Collapsible contracts [Feltey et al. 2018] target specific pathologies for the performance of Typed Racket by carefully merging the contracts generated from types at the boundary between untyped and typed code. Space-efficient manifest contracts [Greenberg 2015] come with a collection of strategies that restrict the accumulation of proxies around a value by removing them — some strategies preserve the meaning of programs, while others do not. Monotonic references [Siek et al. 2015b] builds on space-efficient gradual typing and threesomes to reduce the runtime performance of gradually typed heaps. In all these cases, we conjecture vigilance offers a way to examine the guarantees when these cost-saving approaches are applied to Natural gradual typing. In particular, vigilance can validate the enforcement of the casts resulting from each strategy is sufficient to enforce the same typing histories.

Applying vigilance to approaches that come with advanced typing features may require some expertise and care. For instance, for a gradually typed language with polymorphism, the core structure of the logical relation for vigilance can follow from well understood semantic models of polymorphism [Ahmed 2006]. However, due to the intricate interaction between type dynamic and polymorphism [Ahmed et al. 2017; Igarashi et al. 2017; Toro et al. 2019], the parts of the logical relation of vigilance that are related to typing histories will require redesign and extensions. We leave demonstrating vigilance for systems such as gradual polymorphism as future work.

Other Properties. Type soundness, the mainstay of statically typed languages, has seen numerous interpretations in the gradually typed world. §2 discusses two different type soundness theorems and their shortcomings: the standard type soundness and tag soundness, but more interpretations exist in the literature. Chaudhuri et al. [2017] prove type soundness but only for fully annotated GTL programs. Muehlboeck and Tate [2017] prove a type soundness theorem for a restrictive nominal gradual type system rather than a typical structural gradual type system. Tobin-Hochstadt and Felleisen [2006]'s type soundness concerns a migratory setting where the components of a GTL program have either all or none of their annotations. Vitousek et al. [2017] establish an "open-world" type soundness theorem for a GTL with Transient semantics that guarantees a well-typed program produces either a tag-typed result or certain run-time errors. These properties are all variants of syntactic type soundness. Vigilance is a semantic property that goes beyond type soundness.

Siek et al. [2015a] propose the *gradual guarantees* as a standard for gradually typed languages. Even though the gradual guarantees are useful guidelines for language designers, they are orthogonal to the question of whether the translation-and-semantics combination is a good match for a gradual type system. The static gradual guarantee concerns only the static type system. The dynamic one can be true for a translation and semantics combination that enforces no types at

all. From the languages we examine in this paper, the one with the truer type system violates the static gradual guarantee. The example in Fig. 18 demonstrates why. The return type annotation for function `foo` allows the truer type system to accept that the components of `foo`'s argument are natural numbers; it expects that the translation will insert and the semantics will perform the corresponding tag checks. As a result, given that function `bar` simply returns the result of applying `foo` to its argument, the example type checks. However, if we were to change the return annotation of `foo` to $*$, the truer type system would not be able to decide that the result of `foo` is a pair of naturals, and the example would not type check. Fundamentally, truer typing is flow sensitive and relies on the propagation of "type facts" derived from type annotations through the control flow of a program. Hence, changing the type annotations may affect the outcome of type checking drastically. That said, truer typing is an exercise that demonstrates the power of vigilance — we leave concerns about the gradual guarantees to future work.

$$\mathsf{let\ foo} = \lambda(x{:}*\times*) \rightarrow \mathsf{Nat}\times\mathsf{Nat}.\ \langle \mathsf{fst}\, x, \mathsf{snd}\, x\rangle$$
$$\mathsf{let\ bar} = \lambda(x{:}*\times*) \rightarrow \mathsf{Nat}\times\mathsf{Nat}.\ \mathsf{foo}\ x$$

Fig. 18. Truer: Counterex. to static gradual guarantee

Gradual Type Theory [New et al. 2019, 2021] axiomatizes the dynamic gradual guarantee and a set of contextual equivalence properties as the essence of a well-designed gradually typed language. They show that only a GTL with a simple type system and Natural semantics lives up to this standard, while vigilance shows that even outside this combination, language designers can still rely on type annotations to make decisions about their type system, enforcement, and tooling.

Jacobs et al. [2021] propose an alternative to the gradual guarantees that requires that the embedding of a fully statically typed subset of the GTL into the GTL be fully abstract. We leave it to future work to investigate if vigilance has any connection to properties of this embedding.

Abstracting Gradual Typing (AGT) [Garcia et al. 2016] does not propose a new property but is a method for obtaining a well-designed gradually typed language from a typed one.

There is a significant body of work on equipping gradual type systems with blame in a correct manner [Ahmed et al. 2011; Greenman et al. 2023, 2019a; Vitousek et al. 2017; Wadler and Findler 2009]. Vigilance currently says nothing about blame. Vigilance is concerned with a semantics of types, both static in the form of the typing system and dynamic in the form of boundary annotations. Blame is instead concerned with the mechanisms for providing error messages that developers can use in debugging [Lazarek et al. 2021, 2023]. We conjecture that with additional instrumentation in our logical relation that requires errors at locations informed by typing histories instead of generic errors, we can incorporate properties about blame, but leave that as future work.

Conclusion. Vigilance is a semantic property that describes a gradual type system as three components that work together to validate and recover incomplete type information. When the statics relies on the translation and the dynamics, but the latter two do not deliver, the meaning of types becomes misleading, making type-based reasoning principles faulty. When the translation and the dynamics offer more than what the statics can capture, there is a missed opportunity to increase the strength of the statics, or decrease the amount of dynamic type checks. Vigilance is a compass for exploring the design space, finding adequate design points, and hence, identifying opportunities to incorporate strong reasoning principles into the design of gradual type systems.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation (NSF awards CCF-1910522 and CCF-2237984) and the Defense Advanced Research Projects Agency (DARPA) under Contract No. N66001-21-C-4023. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

REFERENCES

Amal Ahmed. 2006. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming*. Springer, 69–83. https://doi.org/10.1007/11693024_6

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* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 201–214. https://doi.org/10.1145/1926385.1926409

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. https://doi.org/10.1145/3110283

Amal Jamil Ahmed. 2004. *Semantics of Types for Mutable State*. Ph. D. Dissertation. USA. https://dl.acm.org/doi/10.5555/1037736

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *ACM Trans. Program. Lang. Syst.* 23, 5 (sep 2001), 657–683. https://doi.org/10.1145/504709.504712

Felipe Bañados Schwerter, Alison M. Clark, Khurram A. Jafery, and Ronald Garcia. 2021. Abstracting Gradual Typing Moving Forward: Precise and Space-Efficient. *Proc. ACM Program. Lang.* 5, POPL, Article 61 (jan 2021), 28 pages. https://doi.org/10.1145/3434342

Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levy. 2017. Fast and Precise Type Checking for JavaScript. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 56:1–56:30. https://doi.org/10.1145/3133872

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *European Symposium on Programming*. https://doi.org/10.1007/978-3-642-28869-2_11

Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. Collapsible Contracts: Fixing a Pathology of Gradual Typing. 2, OOPSLA (2018), 133:1–133:27. https://doi.org/10.1145/3276503

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. 429–442. https://doi.org/10.1145/2837614.2837670

Michael Greenberg. 2015. Space-Efficient Manifest Contracts. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 181–194. https://doi.org/10.1145/2676726.2676967

Ben Greenman. 2022. Deep and shallow types for gradual languages. In *PLDI*. 580–593. https://doi.org/10.1145/3519939.3523430

Ben Greenman, Christos Dimoulas, and Matthias Felleisen. 2023. Typed–Untyped Interactions: A Comparative Analysis. *ACM Trans. Program. Lang. Syst.* 45, 1, Article 4 (mar 2023), 54 pages. https://doi.org/10.1145/3579833

Ben Greenman and Matthias Felleisen. 2018. A Spectrum of Type Soundness and Performance. 2, ICFP (2018), 71:1–71:32. https://doi.org/10.1145/3235045

Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2019a. Complete Monitors for Gradual Types. *PACMPL* 3, OOPSLA (2019), 122:1–122:29. https://doi.org/10.1145/3360548

Ben Greenman, Lukas Lazarek, Christos Dimoulas, and Matthias Felleisen. 2022. A Transient Semantics for Typed Racket. *Art Sci. Eng. Program.* 6, 2 (2022), 9. https://doi.org/10.22152/programming-journal.org/2022/6/9

Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. 2019b. How to Evaluate the Performance of Gradual Typing Systems. 29, e4 (2019). https://doi.org/10.1017/S0956796818000217

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher Order Symbol. Comput.* 23, 2 (jun 2010), 167–189. https://doi.org/10.1007/s10990-011-9066-z

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On Polymorphic Gradual Typing. *Proc. ACM Program. Lang.* 1, ICFP (2017), 40:1–40:29. https://doi.org/10.1145/3110284

Koen Jacobs, Amin Timany, and Dominique Devriese. 2021. Fully Abstract from Static to Gradual. *Proc. ACM Program. Lang.* 5, POPL, Article 7 (jan 2021), 30 pages. https://doi.org/10.1145/3434288

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. Toward Efficient Gradual Typing for Structural Types via Coercions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 517–532. https://doi.org/10.1145/3314221.3314627

Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2021. How to Evaluate Blame for Gradual Types. 5, ICFP (2021), 68:1–68:29. https://doi.org/10.1145/3473573

Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2023. How to Evaluate Blame for Gradual Types, Part 2. 7, ICFP (2023), 159–186. https://doi.org/10.1145/3607836

Jacob Matthews and Robert Bruce Findler. 2009. Operational Semantics for Multi-Language Programs. *ACM Trans. Program. Lang. Syst.* 31, 3, Article 12 (apr 2009), 44 pages. https://doi.org/10.1145/1498926.1498930

Fabian Muehlboeck and Ross Tate. 2017. Sound Gradual Typing is Nominally Alive and Well. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 56:1–56:30. https://doi.org/10.1145/3133880

Max S. New, Daniel R. Licata, and Amal Ahmed. 2019. Gradual Type Theory. 3, POPL (2019), 15:1 — 15:31. https://doi.org/10.1145/3290328

Max S. New, Daniel R. Licata, and Amal Ahmed. 2021. Gradual type theory. *Journal of Functional Programming* 31 (2021), e21. https://doi.org/10.1017/S0956796821000125

Susan Owicki and David Gries. 1976. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM* 19, 5 (may 1976), 279–285. https://doi.org/10.1145/360051.360224

Andrew Pitts and Ian Stark. 1998. Operational Reasoning for Functions with Local State. In *Higher Order Operational Techniques in Semantics*, Andrew Gordon and Andrew Pitts (Eds.). Publications of the Newton Institute, Cambridge University Press, 227–273. http://www.inf.ed.ac.uk/~stark/operfl.html

Uday S. Reddy and Hongseok Yang. 2003. Correctness of Data Representations Involving Heap Data Structures. In *Programming Languages and Systems*, Pierpaolo Degano (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 223–237. https://dl.acm.org/doi/10.5555/1765712.1765730

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Proceedings of the 2006 Workshop on Scheme and Functional Programming Workshop*. 81–92. http://scheme2006.cs.uchicago.edu/13-siek.pdf

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015a. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

Jeremy G Siek, Michael M Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015b. Monotonic references for efficient gradual typing. In *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 24*. Springer, 432–456. https://doi.org/10.1007/978-3-662-46669-8_18

Jeremy G. Siek and Philip Wadler. 2010. Threesomes, with and without blame. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 365–376. https://doi.org/10.1145/1706299.1706342

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: from Scripts to Programs. In *Dynamic Languages Symposium*. 964–974. https://doi.org/10.1145/1176617.1176755

Matías Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual Parametricty, Revisited. *Proc. ACM Program. Lang.* 3, POPL (2019), 17:1–17:30. https://doi.org/10.1145/3290330

Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. 2019. Optimizing and Evaluating Transient Gradual Typing. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages* (Athens, Greece) *(DLS 2019)*. Association for Computing Machinery, New York, NY, USA, 28–41. https://doi.org/10.1145/3359619.3359742

Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big Types in Little Runtime: Open-World Soundness and Collaborative Blame for Gradual Type Systems. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 762–774. https://doi.org/10.1145/3009837.3009849

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009* (York, UK) *(ESOP '09)*. Springer-Verlag, Berlin, Heidelberg, 1–16. https://doi.org/10.1007/978-3-642-00590-9_1

Received 20-OCT-2023; accepted 2024-02-24