Type Vigilance and the Truth About Transient Gradual Typing

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

Gradually typed languages (GTLs) permit statically and dynamically typed code to coexist in a single language and allow programmers to add type annotations to enforce more precise types on less precisely typed code. But different gradual languages adopt different semantics for checking types dynamically. Two GTLs can have identical type systems and yet provide different type-based guarantees. For example, a GTL with Natural semantics turns type casts into type-enforcing proxies; while one with Transient semantics does simple tag checks when it evaluates type casts and elimination forms. Type soundness does offer a way to distinguish between the run-time guarantees provided by Natural and Transient: for Transient, a simple gradual type system can predict just the tag of a well-typed program, whereas for Natural, it can predict the full type. However, type soundness fails to capture the full extent of the semantic consequences of a choice between Natural and Transient given a GTL and its type system. Specifically, type soundness cannot guarantee to developers that all the type annotations they add to their code are indeed enforced (statically or dynamically) after they are transformed to explicit casts.

We present *vigilance*, a semantic property, that determines whether the semantics performs all the type checks the GTL's type system assumes correct without statically checking, and which hence, become explicit casts. Technically, vigilance asks if a semantics enforces the complete *run-time typing history* of a value, which is made up of all the types that casts are supposed to enforce on the value at run time. We confirm that Natural is *vigilant* for a simple type system; while Transient is *vigilant* for a tag type system, hence, strengthening the conclusions of previous (syntactic) investigation of the two semantics. However, using vigilance as a guide, we are also able to develop a new gradual type system for Transient, dubbed *truer*, that more faithfully reflects the type-level reasoning about programs that is possible in Transient.

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

1 CAN DEVELOPERS RELY ON GRADUAL TYPES?

In gradually typed languages, even sound ones, developers cannot always take type annotations at face value. Since gradual type systems have to accommodate source programs without all the annotations that a typical type system would expect, they make a best effort attempt to type check programs. To make up for this possibly partial static type checking, gradually typed languages (GTLs) aim to detect dynamically whether a program's behavior and its existing annotations are at odds. One way to describe the behavior of a gradually typed program is with a translation to an *internal gradually typed language* (IGTL) with *type casts* and *type assertions*. These are supposed to perform checks at run time that bridge the gap between what the type system of the GTL can deduce on its own and what the developer claims through the type annotations of the program. Hence, developers should still be able to rely on the type annotations in a source program as they reason about their code; if there is a mismatch between an annotation and the program's behavior, the type casts and assertions of the translated program should reveal it by raising a dynamic type error. Unfortunately, there is a problem: in the name of efficiency and ease of implementation, the evaluation of a translated program may take place in an IGTL that does not perform all the checks needed by the type system of the GTL. In other words, the semantics of the IGTL may not be able to fill in all the type-level facts that the type system of the GTL assumed would be established at run time.

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

This paper develops a formal framework that determines ***whether the type system of a GTL and the semantics of an IGTL are a good match for each other***. On one hand, given a type system for a GTL, the framework of the paper examines whether the semantics of an IGTL can mislead developers who assume that the language does its due diligence to enforce types as the GTL type system promises. On the other hand, given a semantics for an IGTL, the framework guides the design of a type system for a GTL such that the IGTL semantics and the GTL type system work together to truly enforce the developer's type annotations.

To make the discussion concrete, consider Reticulated Python, a gradually typed variant of Python [19]. Notably, Reticulated Python has two back ends: Natural Reticulated relies on proxies that are difficult (if not impossible) to implement in a performant manner, while Transient Reticulated offers a light-weight alternative. That is, the behavior of a Reticulated Python program can be described through its translation and evaluation into two IGTLs that both aim to enforce the types of Reticulated programs but one has Natural and the other Transient semantics.

```
def rolling_hash(el):
    ...
def encode(file_path, word_coder: Function([String],Int)):
    ...
encode("~\oopsla23\paper.tex",rolling_hash)
```

Fig. 1. Dynamically Typed Argument for a Typed Parameter

The example in Figure 1 demonstrates the workings of Natural Reticulated Python. Here, the developer applies `encode`, which expects as its second argument a function from strings to integers, to `rolling_hash`, which has no type annotations. Even though Reticulated Python's type checker does not have at its disposal the necessary annotations to derive that `rolling_hash` indeed has type `Function([String],Int)`, it accepts the program as well-typed. To compensate for the partial type checking, the translation of the call to `encode` at the bottom of the snippet injects a cast around `rolling_hash` from the dynamic type `Dyn`[1] to the expected type `Function([String],Int)`. The role of the cast is to check whether `rolling_hash` behaves as a function from strings to integers whenever it is called by the body of `encode`. Consequently, in Natural, the cast checks that `rolling_hash` is a function and then wraps it in a proxy that, in turn, checks that the results of calls to `rolling_hash` are integers.

The same code evaluates differently in Transient Reticulated Python. As before, the translation injects a cast from type `Dyn` to type `Function([String],Int)` around `rolling_hash`. However, the Transient cast only checks that `rolling_hash` is a function and does not create a proxy. To counter for the absence of the proxy, the translation further re-writes the body of `encode` and injects type assertions to ensure that the results of any calls to `world_coder` are integers. In other words, Transient performs lightweight tag checks instead of wrapping values in proxies, but seems to establish the same type-level facts for a program as Natural with its costly proxies.

However, the above conclusion is misleading. This becomes clear when developers attempt to write dynamically typed code that uses a function with type other than `Dyn`. This situation arises ubiquitously in gradually typed languages [8, 9, 23], where developers routinely opt to import libraries with typed interfaces even while writing dynamically typed code, because they count on the typed interfaces for guidance on how to structure their code.

Unfortunately, many libraries with typed interfaces are in fact dynamically typed themselves, and so the interface is nothing more than a thin veneer of possibly incorrect type annotations. The casts and assertions injected by the translation of the developer's code to one of the IGTLs may not perform all the checks that are necessary to validate these annotations, and the developer may

[1]Code without annotations implicitly has type `Dyn`.

only discover that they are incorrect after a painful debugging process; it all depends on whether the semantics of the IGTL enforce the types the developer relied upon.

The variant of our running example in Figure 2 demonstrates this situation. In this example, the developer opts not to add any type annotations to `encode`. Instead the developer uses `make_me_typed`, a hand-rolled type adapter, to get some of the benefits of types while implementing the dynamically typed `encode`. In particular, the developer expects that assigning the result of `make_me_typed` to `word_coder` makes unnecessary the use of defensive code that inspects the results of calls to `word_coder` to determine whether they are integers. However, whether Reticulated Python ends up checking that calls to `word_coder` produce integers depends on the semantics of the IGTL the program translates to. In Natural, `word_coder` has the expected proxy that inspects the results of the function; in Transient, there is no proxy and, since `word_coder` is an identifier of type `Dyn`, the compiler injects no type assertions for the results of calls to `word_coder`. Hence, in Transient, a developer that trusts the annotations of the program and does not code defensively may discover that the results of `word_coder` are not integers when some other part of the code handles them.

```
def make_me_typed(f)->Fun([String],Int):
    return f
def encode(file_path, word_coder):
    word_coder = make_me_typed(word_coder)
    ...
encode("~\oopsla23\paper.tex",rolling_hash)
```

Fig. 2. Applying a Hand-Rolled Type Adapter

Type Soundness is Not Enough. At first glance, this difference between Natural and Transient seems like an issue that type soundness for Reticulated Python should clarify. Type soundness does distinguish between the two [6], but falls short of fully characterizing this difference in their behaviour. Intuitively, in Transient, the use context of a function is expected to inspect the results of the 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, Transient can say little about programs that produce functions. In contrast, in Natural, the proxy around a function enforces the expected type independently of the use context. Formally, the picture becomes a bit more nuanced. If a source program is well-typed at type τ, then, it translates to a Natural program with casts that has type τ. When the translated program runs to a value, the value also has type τ. In contrast, the differently-translated Transient program and its result value have types that match the tag of τ (i.e. its top type constructor), but that are not necessarily exactly equal to τ (tag soundness). Hence, type soundness seems to reveal that the choice of IGTL affects the predictive power of Reticulated Python's type system.

However, type soundness stops short of explaining whether the semantics of either IGTL performs all the checks the Reticulated Python type system, and hence developers, rely on. After all, type soundness only connects the type of the source program with the type of the translated IGTL program's result. Therefore, all annotations that do not contribute to that goal are immaterial. And this is exactly, the situation with the running example. First, given the pervasive lack of annotations in the example, Reticulated Python's type-checker determines that, despite the reassignment step, the type of `word_coder` is `Dyn`, which is inhabited by all values, not just functions from strings to integers. So type soundness says nothing about `word_coder` directly. Second, the developer 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 than `Dyn` for `word_coder`. While the typing rules of Reticulated Python and type soundness do stipulate that the result of `make_me_typed(word_coder)` needs to behave as a function from integers to strings, this is a fact that the type system cannot prove without the help of the semantics of the underlying IGTL, again due to the sparsity of type annotations. And as discussed above, in Natural, which employs a proxy around the result of `make_me_typed(word_coder)`, the expected type is indeed enforced. But, in Transient, which does not create the proxy, the type, and the corresponding annotation, are "forgotten". In general, for Transient, the avoidance of proxies comes at the price of a weaker (tag) soundness guarantee, but in this case, this weaker guarantee is not the reason that the developer cannot use type-level reasoning alone to justify the elision of defensive checks on the results of `word_coder`. In fact, the Forgetful [4] and Amnesic [7] variants of Natural create the same proxy as Natural but then remove it upon the assignment to `word_coder` to reduce the running time and memory cost from proxies. Hence, their net effect is exactly the same as that of Transient in this example but they are as type sound as Natural. The root of the issue is that from the perspective of type soundness, it does not matter whether the annotation on the result of `make_me_typed(word_coder)` is ever enforced. In other words, the developer needs to know something about the semantics of the IGTL beyond type soundness to be able to trust that Reticulated Python is going to enforce the annotation.

Vigilance: Type Annotations You Can Trust. The key insight of this paper is that a GTL that developers can trust should enforce, with the help of its target IGTL, all type annotations in the code that are relevant to the behavior of the code. Intuitively, a GTL should enable developers to reliably take into account type annotations when they reason about code, in the same way that they rely on type annotations when they reason about code in strongly typed languages, or the way they rely on manually placed type-level predicates in dynamically typed languages. Returning to the example, Reticulated Python, under either Natural or Transient, should empower the developer to deduce that `make_me_typed(word_coder)` behaves according to the return type of `make_me_typed`, and so does `word_coder` after the re-assignment.

To formally capture this intuition, this paper develops a new semantic property called *vigilance*. The goal of vigilance is to describe how the semantics of an IGTL impacts the enforcement of the type annotations in a GTL program.

To that end, our formal development starts with the definition of syntax for a GTL based on the gradually typed λ-calculus [14]. For each type system that we wish to consider, we define a single type-preserving translation that maps well-typed (under the relevant type system) programs in this GTL to programs in a family of IGTLs that share a syntax and type system but differ in their reduction semantics. This creates the setting for an apples-to-apples comparison of how different IGTL semantics enforce a particular type system. Since the translation is type preserving we reduce the question of the relation between the semantics of the IGTLs and the enforcement of GTL types to the comparison of models of IGTL types under different IGTL semantics.

Hence, after equipping our IGTL family with two different reduction semantics, for Natural and Transient respectively, we define vigilance using a (step-indexed) unary logical relation that models IGTL types τ as sets of values v that inhabit them. In contrast to a typical logical relation for type soundness [1], vigilance comes with an extra index, the *type history* Ψ, which is a log that collects the types present on each cast or assertion in the program that a particular value has passed through. Semantic type soundness says that a language is semantically type sound if and only if any well-typed expression $e : \tau$ "behaves like" τ. We say that the semantics of an IGTL are *vigilant* for a type system if this remains true when the latter constraint is strengthened to say that in any well-formed type history Ψ (capturing potential casts and assertions from a context), e behaves like τ, by which we mean that, if evaluating e produces a value v (as well as a potentially-larger type history Ψ', capturing casts and assertions present in e), then v not only behaves like τ (in the conventional sense), but also like all of the types in $\Psi'(v)$.

Back to the running example, vigilance communicates to the developer the difference between Natural and Transient Reticulated Python regarding the guarantees each offers about the type annotations in the code from Figure 2. In order for either Natural or Transient to be vigilant for the simple type system of Reticulated Python, the value bound to `word_coder` after the re-assignment should behave not only according to type `Dyn`, but also, due to the type history, according to type `Fun([String],Int)`. Only Natural meets this standard, and only in Natural can the developer rely on the type annotation of `make_me_typed` to avoid programming encode defensively.

Besides informing developers, vigilance is a guiding tool for language designers. To demonstrate this aspect of vigilance, the paper revisits Transient and asks for what type system is Transient vigilant. An initial answer, which confirms prior work on Transient and tag soundness, is that one such type system maps expressions to a type tag rather than a type. Hence, this *tag type system* accepts programs that have severely imprecise types, which makes plain the difference between what Reticulated Python's simple type system promises and what its Transient semantics achieves.

However, the fact that Transient is vigilant for tag typing, also reveals that Transient can provide strong support when developers reason about Reticulated Python code. For example, consider the code in Figure 3. Here, a developer 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. Similarly to the previous examples, the developer creates a typed interface for this library that restricts it to `PNG`s. Unlike before, in this example, the typed interface does a bit more than just acting as a veneer of types; it uses `crop` to reduce the sizes of the pair of images that `segment` produces. If the example is evaluated in Transient, due to vigilance, a developer can rely on the return type annotation for `segment_png_small`: vigilance for tag typing guarantees that Transient checks that the results of the calls to `png_small` in the body of `segment_png_small` are `PNG`s, and hence, the result of `segment_png_small` has type `(PNG, PNG)`.

```
# Typed interface
def png_crop(img : PNG) -> PNG:
    return crop(png)
def segment_png_small(img : PNG) -> (PNG, PNG):
    let (a, b) = segment(img)
    in png_crop(a), png_crop(b)

let foreground = segment_png_small(my_png)[0]
in write_png(foreground, "fg.png")
```

Fig. 3. Using a Type Adapter for an Image Library

Vigilance: Towards Truer Types. More interesting is the fact that vigilance can also point the way for designing a new gradual type system for which Transient is vigilant, and that maps expressions to more precise types than the tag type system. In detail, this *truer type system* makes limited use of union and intersection types in order to reflect the outcomes of Transient casts and assertions to the types of expressions, similarly to the way a developer can use these tag checks to reason about code as we discuss above. A consequence of this more precise type system is that some of the checks that are necessary so that Transient is vigilant for the tag type system can be elided in a provably correct manner when pairing Transient 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` has indeed a type that matches precisely 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`.

The following table summarizes the results of the paper. Each cell of the table corresponds to a pairing of a semantics and a type system. When the cell contains a ✓, that semantics is vigilant for that type system; otherwise it is not.

	Simple Typing	Tag Typing	Truer Typing
Natural	✓	×	×
Transient	×	✓	✓

In addition, to the results discussed above, the table includes two negative results about Natural and the tag and truer type systems. It turns out that the simple type system rejects some programs that the other two accept, and these program expose that tag and truer typing are not type sound for Natural. Since vigilance strengthens type soundness, Natural is also not vigilant for the two type systems.

Outline. The remainder of the paper is organized as follows. §2 places vigilance in the context of prior work on the correctness properties for gradual type systems. §3 presents the formal linguistic setting of the paper and §4 builds on that to define vigilance and proves that Natural is vigilant for simple typing. §5 develops the truer type system, shows that Transient is vigilant for tag typing and truer typing, and proves that truer renders unnecessary some of the checks that Transient performs. §6 discusses future directions and concludes.

2 VIGILANCE AND PRIOR RESEARCH ON GRADUAL TYPING

The question of what properties should be used to characterize gradually-typed languages has been a research theme since the early days of gradual typing. Over the last 15 years, different properties have been proposed, each approaching the correctness of gradual typing from different perspectives. In this context, vigilance is a new semantic property that no existing property subsumes.

Type soundness, the mainstay of statically typed languages, can also be used to categorize gradual type systems. However, many different interpretations have emerged under the same name. §1 discussed two different type soundness theorems and their shortcomings in the context of Reticulated Python: the standard type soundness and tag soundness. But the heterogeneity of how the literature uses the term type soundness goes well beyond that. Chaudhuri et al. [2] prove standard type soundness but only for fully annotated GTL programs. Muehlboeck and Tate [11] prove a type soundness theorem for a restrictive nominal gradual type system rather than a typical structural gradual type system. Tobin-Hochstadt and Felleisen [16]'s type soundness concerns a migratory setting where the components of a GTL program have either all their annotations or no annotations. Furthermore, they strengthen type soundness to a *blame theorem* [10, 16, 17, 22] that describes what kinds of run time type errors a well-typed program can raise. Vitousek et al. [21] establish an "open-world" type soundness theorem for a GTL with Transient semantics that makes no predictions about the evaluation of a well-typed program except what run-time errors can occur, similar to the error-reporting part of the blame theorem. Its open-world nature is due to the fact that its (weak) guarantees extend to well-typed programs placed in arbitrary contexts. These properties are all variants of syntactic type soundness. Vigilance is a semantic property that goes beyond (semantic) type soundness by requiring that the semantics of an IGTL enforce the run-time typing history of any value computed during the evaluation of a program. The language framework that we use to develop vigilance is inspired by that of Greenman and Felleisen [6], who show how a framework that irons away the differences between different GTLs enables an apples-to-apples comparison of their semantics.

Complete monitoring [7] is another direct inspiration for vigilance. It is a syntactic property that stipulates that whenever a value crosses into a new context by passing through a cast, the semantics should either perform some checks, or break up the cast into new (simpler) casts. Greenman et al. [7] use complete monitoring, along with blame soundness and completeness, and observations about how semantics fit into an error preorder to distinguish existing semantics from the literature. Much like vigilance, complete monitoring ensures the semantics has the opportunity to perform

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

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

checks at all the necessary places to enforce types, but, unlike vigilance, does not ensure that those checks enforce the semantics of those types.

Siek et al. [15] propose the *gradual guarantee* to prescribe how increasing the precision of the type annotations in a GTL program should affect type checking (static) and program evaluation (dynamic). Even though the gradual guarantee is an important and useful guideline for language designers, it is orthogonal to the question of whether the semantics of an IGTL enforce the type annotations of a GTL program. The static portion of the gradual guarantee concerns only the type system. The dynamic portion can be true for a semantics that enforces no types at all.

Gradual Type Theory [12] axiomatizes the dynamic gradual guarantee and a set of contextual equivalence properties as the essential properties of a well-designed gradually typed language. In particular, the equivalence properties entail that developers can reason in the gradually typed setting using the same basic principles as in the typed setting. They show that only a GTL with a simple type system and Natural semantics lives up to this standard. Our study of vigilance shows that even when a GTL does not preserve all the reasoning properties of simple types, developers can still rely on type annotations to make decisions about how to structure their code, if the type system and the semantics are in sync.

Abstracting Gradual Typing (AGT) [3] does not propose a new property but is a method for obtaining a well-designed gradually typed language from a typed one. Their system produces IGTLs that manage "evidence objects", which act as "proofs" that the semantics enforce a value's typing history. Hence, we conjecture that AGT is a recipe for creating semantics for an IGTL that are by construction vigilant for the type system of the GTL. A novelty of this paper is that we also do the converse: start with the semantics of an IGTL (Transient) and arrive at a type system for a GTL (truer) for which it is vigilant.

3 FROM A GTL TO A FAMILY OF IGTLS

The top portion of Figure 4 presents the syntax of λ^{GTL}, our GTL, which is inspired by the gradually-typed λ-calculus [14]. Most of its features 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 — matching the way that Reticulated Python developers can annotate function definitions. 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, similar to Reticulated Python, are translated to an IGTL, before delving into the type checking and evaluation of λ^{GTL} expressions, we discuss briefly the syntax of λ^{IGTL}, our family of IGTLs. The bottom portion of Figure 4 shows the syntax of λ^{IGTL} 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,

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

$$\frac{\Gamma,\ (x{:}\tau) \vdash_{\mathsf{sim}} t : \tau'' \rightsquigarrow e}{\begin{array}{c}\Gamma \vdash_{\mathsf{sim}} \lambda(x{:}\tau) \longrightarrow \tau'.\, t : \tau \longrightarrow \tau' \\ \rightsquigarrow \lambda(x{:}\tau).\,([\tau' \swarrow \tau'']e)\end{array}}
\qquad
\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{sim}} t_1 : \tau \longrightarrow \tau' \rightsquigarrow e_1 \\ \Gamma \vdash_{\mathsf{sim}} t_2 : \tau'' \rightsquigarrow e_2\end{array}}{\begin{array}{c}\Gamma \vdash_{\mathsf{sim}} t_1\ t_2 : \tau' \\ \rightsquigarrow \mathsf{app}\{\tau'\}\, e_1\ ([\tau \swarrow \tau'']e_2)\end{array}}
\qquad
\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{sim}} t_1 : * \rightsquigarrow e_1 \\ \Gamma \vdash_{\mathsf{sim}} t_2 : \tau' \rightsquigarrow e_2\end{array}}{\begin{array}{c}\Gamma \vdash_{\mathsf{sim}} t_1\ t_2 : * \\ \rightsquigarrow \mathsf{app}\{*\}\ (\mathsf{cast}\ \{* \longrightarrow * \Leftarrow *\}\ e_1)\ e_2\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{sim}} t_b : \mathsf{Bool} \rightsquigarrow e_b \\ \Gamma \vdash_{\mathsf{sim}} t_1 : \tau_1 \rightsquigarrow e_1 \\ \Gamma \vdash_{\mathsf{sim}} t_2 : \tau_2 \rightsquigarrow e_2\end{array}}{\begin{array}{c}\Gamma \vdash_{\mathsf{sim}} \mathsf{if}\ t_b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \tau_1 \sqcup_{\leqslant} \tau_2 \\ \rightsquigarrow \mathsf{if}\ e_b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\end{array}}
\qquad
\begin{array}{c}
[\tau \swarrow \tau']e = \begin{cases} e & \text{if } \tau \leqslant: \tau' \\ \mathsf{cast}\ \{\tau \Leftarrow \tau'\}\ e & \text{if } \tau \not\leqslant: \tau' \wedge \tau \sim \tau' \end{cases} \\[2ex]
\tau_1 \sqcup_{\leqslant} \tau_2 = \begin{cases} \tau_1 & \text{if } \tau_2 \leqslant: \tau_1 \\ \tau_2 & \text{if } \tau_1 \leqslant: \tau_2 \end{cases}
\end{array}$$

Fig. 5. Translating λ^{GTL} to λ^{IGTL}

pair projections and function applications also have type annotations. Third, λ^{IGTL} 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^{IGTL}, 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 GTL. Hence, the translation injects type assertions and casts exactly at these spots.

Figure 5 presents type checking for λ^{GTL} expressions t and their translation to λ^{IGTL} expressions e with a single judgment $\Gamma \vdash_{\mathsf{sim}} t : \tau \rightsquigarrow e$ — the sim annotation indicates simple typing to distinguish it from the tag and tru type systems we present later. A λ^{GTL} function type checks at its type annotation $\tau \longrightarrow \tau'$ if its body type checks at some type τ''. To bridge the potential gap between τ'' and τ', the translation of the function produces a λ^{IGTL} 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 but not a subtype of τ' (both subtyping and compatibility are standard). Conditionals type check and translate in a straightforward and recursive manner. The type of the conditional is the supertype between the types of its two branches. Translated applications obtain ascriptions 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.

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

$$t = (\lambda(f{:}\mathsf{Nat} \longrightarrow \mathsf{Nat}) \longrightarrow *.\, f\ 42)\ \lambda(x{:}*) \longrightarrow *.\, x$$

and its λ^{IGTL} image

$$e = \mathsf{app}\{*\}\ (\lambda f : \mathsf{Nat} \longrightarrow \mathsf{Nat}.\ \mathsf{cast}\ \{* \Leftarrow \mathsf{Nat}\}\ \mathsf{app}\{\mathsf{Nat}\}\ f\ 42)\ \mathsf{cast}\ \{\mathsf{Nat} \longrightarrow \mathsf{Nat} \Leftarrow * \longrightarrow *\}\ \lambda x : *.\, x.$$

The example is reminiscent of the one from Figure 1; it involves a higher-order function whose parameter f has an annotation. Hence, the translation injects in e a cast to ensure that the argument of the higher-order function behaves as prescribed by the annotation. The translation also introduces a trivial cast around the body of the higher-order function to bridge the gap between Nat, the deduced type for the body of the function, and $*$, the expected type according to the type annotation in t. Conversely, the translation does not introduce a cast around 42, the argument to f in the body of the higher-order function, as its deduced and expected type (Nat) are the same. Finally, the translation annotates all applications with their expected types, affirming the return types from the type annotation in t.

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

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

Fig. 6. The Type System of λ^{IGTL}

Figure 6 gives some rules for the typing judgment for λ^{IGTL}: $\Gamma \vdash_{\mathsf{sim}} e : \tau$. In general, the type system of λ^{IGTL} is straightforward and closely follows that of λ^{GTL}.

The translation has a key property: it maps well-typed λ^{GTL} expressions to well-typed λ^{IGTL} expressions with the same type.

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

As discussed in §1, type preservation allows us to focus on λ^{IGTL} to determine the impact of Natural and Transient on the enforcement of the types of a well typed λ^{GTL} expression t. After all, if a semantics enforces all the types that the type system of λ^{IGTL} promises for the translated (and well typed) expression e, then the type preservation of the translation implies that the semantics also enforces all the types promised by the type system of λ^{GTL} for t.
Note: The complete formal development of λ^{GTL} and λ^{IGTL} along with all the definitions and proofs of theorems from the paper are in the supplemental material.

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

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 IGTL 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^{IGTL}. 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^{IGTL}. This reduction 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 semantics populates 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^{IGTL} 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^{IGTL} while leaving the rest of the formalism unchanged. The top left of Figure 7 depicts these extensions. The *monitor* expression $\mathsf{mon}\,\{\tau \Leftarrow \tau\}\ e$ regulates the evaluation of cast expressions; it is an intermediate term 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. There are two kinds of errors in the evaluation language of λ^{IGTL}: $\mathsf{Err}^{\bullet}$ 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^{IGTL} 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 N (for Natural) or T (for Transient). The only difference between the two notions of reduction is in their *compatibility* metafunction $\sim^L_c$, 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 $v \sim \lfloor\tau\rfloor$ that checks whether v matches the *tag* $\lfloor\tau\rfloor$ of the given type τ. Put differently, $v \sim^L_c \tau$ 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^{IGTL} construct that triggers a possible tag check and the particular semantics of λ^{IGTL}. 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 Figure 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 — Transient performs such tag tests to "protect" the bodies of functions from arguments of the wrong type in the absence of proxies. When the compatibility metafunction returns True the evaluation proceeds with a beta-reduction; otherwise it raises a TypeErr, i.e., a dynamic type error. 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 where an application does not involve a function is one of the cases that the type system of λ^{IGTL} should prevent. Hence, the corresponding reduction rule raises Wrong to distinguish this unexpected error from dynamic type errors.

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^{IGTL} relies on them. Specifically, a monitor expression performs any checks a proxy would perform using the compatibility metafunction, and produces a fresh label to record in the value log that after the fresh label is associated 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.

The sequence of reductions in Figure 8 gives a taste of the log-based semantics of λ^{IGTL} through the evaluation of the example expression e from above. The reduction sequence is the same for both Natural and Transient except for the step marked with (†). Up to that point, both semantics store intermediate values in the value log Σ, check with a cast that ℓ_2 points to a function, and, after the check succeeds, create a new label ℓ_3 that the updated Σ associates with the types from the cast. For step (†), both semantics perform a beta-reduction. But via the compatibility metafunction, Transient also checks that the argument of ℓ_1 is indeed a function. The two semantics get out of sync again after the last step of the shown reduction sequence. Specifically, for the remainder of

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

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

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

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

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

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

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

$$\begin{array}{l} \Sigma, \ell_0\ \ell_1 \hookrightarrow_L \Sigma, \mathsf{TypeErr}(\tau_1, \ell_1) \\ \quad \text{if } \Sigma(\ell_0) = (\lambda(x_0\!:\!\tau_1).\, e_0, \_) \\ \quad \text{and } \neg\mathsf{pointsto}(\Sigma, \ell_1) \sim^L_{\mathit{assert}} \tau_1 \end{array}$$

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

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

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

$$\begin{array}{l} \Sigma, \mathsf{cast}\,\{\tau_1 \Leftarrow \tau_2\}\, \ell_0 \\ \hookrightarrow_L \Sigma, \mathsf{mon}\,\{\tau_1 \Leftarrow \tau_2\}\, \ell_0 \\ \quad \text{if } \mathsf{pointsto}(\Sigma, \ell_0) \sim^L_{\mathit{cast}} \tau_1 \\ \quad \text{and } \mathsf{pointsto}(\Sigma, \ell_0) \sim^L_{\mathit{cast}} \tau_0 \end{array}$$

$$\begin{array}{l} \Sigma, \mathsf{cast}\,\{\tau_1 \Leftarrow \tau_2\}\, \ell_0 \\ \hookrightarrow_L \Sigma, \mathsf{TypeErr}(\tau_1, \ell_0) \\ \quad \text{if } \neg\mathsf{pointsto}(\Sigma, \ell_0) \sim^L_{\mathit{cast}} \tau_1 \end{array}$$

$$\begin{array}{l} \Sigma, \mathsf{mon}\,\{\tau_1 \Leftarrow \tau_2\}\, \ell_0 \\ \hookrightarrow_L \Sigma[\ell_1 \mapsto (\ell_0, \mathsf{some}(\tau_1, \tau_2))], \ell_1 \\ \quad \text{if } \ell_1 \notin \mathit{dom}(\Sigma) \\ \quad \text{and } \mathsf{pointsto}(\Sigma, \ell_0) = v \\ \quad \text{where } v = i \text{ or } \mathsf{True} \text{ or } \mathsf{False} \\ \quad \text{and } v \sim^L_{\mathit{mon}} \tau_1 \wedge v \sim^L_{\mathit{mon}} \tau_2 \end{array}$$

$$\begin{array}{l} \Sigma, \mathsf{mon}\,\{\tau_1 \Leftarrow \tau_2\}\, \ell_0 \\ \hookrightarrow_L \langle \mathsf{mon}\,\{\mathit{fst}(\tau_1) \Leftarrow \mathit{fst}(\tau_2)\}\, \ell_1, \\ \qquad \mathsf{mon}\,\{\mathit{snd}(\tau_1) \Leftarrow \mathit{snd}(\tau_2)\}\, \ell_2 \rangle \\ \quad \text{if } \Sigma(\ell_0) = (\langle \ell_1, \ell_2 \rangle, \_) \end{array}$$

$$\begin{array}{l} \Sigma, \mathsf{mon}\,\{\tau_1 \Leftarrow \tau_2\}\, \ell_0 \\ \hookrightarrow_L \Sigma[\ell_1 \mapsto (\ell_0, \mathsf{some}(\tau_1, \tau_2))], \ell_1 \\ \quad \text{if } \ell_1 \notin \mathit{dom}(\Sigma) \\ \quad \text{and } \mathsf{pointsto}(\Sigma, \ell_0) = v \\ \quad \text{and } v = \lambda(x_0\!:\!\tau_1).\, e_0 \\ \quad \text{and } v \sim^L_{\mathit{mon}} \tau_1 \wedge v \sim^L_{\mathit{mon}} \tau_2 \end{array}$$

$$\begin{array}{l} \Sigma, \mathsf{mon}\,\{\tau_1 \Leftarrow \tau_2\}\, \ell_0 \\ \hookrightarrow_L \Sigma, \mathsf{TypeErr}(\tau_1, \ell_0) \\ \quad \text{if } \neg\mathsf{pointsto}(\Sigma, \ell_0) \sim^L_{\mathit{mon}} \tau_1 \end{array}$$

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

c	$v \sim^{\mathsf{N}}_c \tau$	$v \sim^{\mathsf{T}}_c \tau$
cast	$v \sim \lfloor\tau\rfloor$	$v \sim \lfloor\tau\rfloor$
mon	$v \sim \lfloor\tau\rfloor$	True
assert	True	$v \sim \lfloor\tau\rfloor$

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

the evaluation, Natural performs checks due to the monitor expressions such as the ones around ℓ_3 and ℓ_4, while Transient performs the checks stipulated by assert expressions.

4 TYPE VIGILANCE

In this section, we define *vigilance*, a semantic property of an IGTL that says that every value must satisfy *both* the type ascribed to it by the 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 more standard semantic-type-soundness part of our step-indexed logical relation. Figure 9 presents the value and expression relations. Ignoring, for the moment, the highlighted terms in the figure, the value relation $\mathcal{V}^L[\![\tau]\!]$ 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.

$$\begin{array}{ll}
\multicolumn{2}{l}{\emptyset, \mathsf{app}\{*\}\ (\lambda f : \mathsf{Nat} \to \mathsf{Nat}.\, \mathsf{cast}\, \{* \Leftarrow \mathsf{Nat}\}\ \mathsf{app}\{\mathsf{Nat}\}\, f\ 42)\ (\mathsf{cast}\, \{\mathsf{Nat} \to \mathsf{Nat} \Leftarrow * \to *\}\ \lambda x : *.\, x)} \\
\longrightarrow_L^* & \{\ell_1 \mapsto (v_1, \mathsf{none}), \ell_2 \mapsto (v_2, \mathsf{none})\}, \mathsf{app}\{*\}\, \ell_1\ (\mathsf{cast}\, \{\mathsf{Nat} \to \mathsf{Nat} \Leftarrow * \to *\}\ \ell_2) \\
\longrightarrow_L^* & \{\ell_1 \mapsto (v_1, \mathsf{none}), \ell_2 \mapsto (v_2, \mathsf{none})\}, \mathsf{app}\{*\}\, \ell_1\ (\mathsf{mon}\, \{\mathsf{Nat} \to \mathsf{Nat} \Leftarrow * \to *\}\ \ell_2) \\
\longrightarrow_L^* & \{\ell_1 \mapsto (v_1, \mathsf{none}), \ell_2 \mapsto (v_2, \mathsf{none}), \ell_3 \mapsto (l_2, \mathsf{some}(\mathsf{Nat} \to \mathsf{Nat}, * \to *))\}, \mathsf{app}\{*\}\, \ell_1\ \ell_3 \\
\longrightarrow_L^* & \{\ell_1 \mapsto (v_1, \mathsf{none}), \ell_2 \mapsto (v_2, \mathsf{none}), \ell_3 \mapsto (l_2, \mathsf{some}(\mathsf{Nat} \to \mathsf{Nat}, * \to *))\}, \mathsf{assert}\ *\ (\ell_1\ \ell_3) \\
\longrightarrow_L & \{...\}, \mathsf{assert}\ *\ \mathsf{cast}\, \{* \Leftarrow \mathsf{Nat}\}\ \mathsf{app}\{\mathsf{Nat}\}\, \ell_3\ 42 \quad (\dagger) \\
\longrightarrow_L^* & \{..., \ell_4 \mapsto (42, \mathsf{none})\}, \mathsf{assert}\ *\ \mathsf{cast}\, \{* \Leftarrow \mathsf{Nat}\}\ \mathsf{app}\{\mathsf{Nat}\}\, \ell_3\ \ell_4 \\
\longrightarrow_L^* & \{...\}, \mathsf{assert}\ *\ \mathsf{cast}\, \{* \Leftarrow \mathsf{Nat}\}\ \mathsf{assert}\, \mathsf{Nat}\, \mathsf{mon}\, \{\mathsf{Nat} \Leftarrow *\}\ (\ell_3\ (\mathsf{mon}\, \{* \Leftarrow \mathsf{Nat}\}\ \ell_4))
\end{array}$$

Fig. 8. Example of log-based reduction.

$$\begin{array}{rcl}
\mathcal{V}^L[\![C]\!] & \triangleq & \{(k, \Psi, \Sigma, \ell) \mid \mathsf{pointsto}(\Sigma, \ell) \in C\} \\
\mathcal{V}^L[\![\tau_1 \times \tau_2]\!] & \triangleq & \{(k, \Psi, \Sigma, \ell) \mid \Sigma(\ell) = (\langle \ell_1, \ell_2 \rangle, \_) \wedge\ (k, \Psi, \Sigma, \ell_1) \in \mathcal{V}^L[\![\tau_1]\!] \wedge\ (k, \Psi, \Sigma, \ell_2) \in \mathcal{V}^L[\![\tau_2]\!]\} \\
\mathcal{V}^L[\![\tau_1 \to \tau_2]\!] & \triangleq & \{(k, \Psi, \Sigma, \ell) \mid \forall\, (j, \Psi') \sqsupseteq (k, \Psi),\ \Sigma' \supseteq \Sigma,\ \ell_v,\ \tau_0 \geqslant: \tau_2. \\
 & & \qquad \Sigma' : (j, \Psi') \wedge (j, \Psi', \Sigma', \ell_v) \in \mathcal{V}^L[\![\tau_1]\!].\ \Rightarrow (j, \Psi', \Sigma', \mathsf{app}\{\tau_o\}\, \ell\ \ell_v) \in \mathcal{E}^L[\![\tau_0]\!]\} \\
\mathcal{V}^L[\![*]\!] & \triangleq & \{(k, \Psi, \Sigma, \ell) \mid (k-1, \Psi, \Sigma, \ell) \in \mathcal{V}^L[\![C]\!] \cup \mathcal{V}^L[\![* \times *]\!] \cup \mathcal{V}^L[\![* \to *]\!]\} \\
\mathcal{E}^L[\![\tau]\!] & \triangleq & \{(k, \Psi, \Sigma, e) \mid \forall j \leq k.\ \forall \Sigma' \supseteq \Sigma, e'.\ (\Sigma, e) \longrightarrow_L^j (\Sigma', e') \wedge \mathtt{irred}(e') \Rightarrow (e' = \mathsf{Err}^\bullet \vee \\
 & & \qquad (\ \exists (k-j, \Psi') \sqsupseteq (k, \Psi).\ \Sigma' : (k-j, \Psi') \wedge\ (k-j, \Psi', \Sigma', e') \in \mathcal{V}^L[\![\tau]\!]))\}
\end{array}$$

Fig. 9. Vigilance: Value and Expression Relations

For base types, ℓ belongs to the relation $\mathcal{V}^L[\![B]\!]$ 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, $\mathcal{V}^L[\![\tau_1 \times \tau_2]\!]$ contains only labels with 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 $\mathcal{V}^L[\![\tau_1 \to \tau_2]\!]$ 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.

Finally, for the dynamic type, $\mathcal{V}^L[\![*]\!]$ is an untagged union over base types $\mathcal{V}^L[\![B]\!]$, pairs of dynamic types $\mathcal{V}^L[\![* \times *]\!]$, and functions between dynamic types $\mathcal{V}^L[\![* \to *]\!]$. Since these types are not structurally smaller than $*$, step-indexing becomes crucial. For well-foundedness, a term 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 $\mathcal{E}^L[\![\tau]\!]$. 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 $\mathcal{V}^L[\![\tau]\!]$.

The logical relation defined thus far is a mostly standard type soundness relation. 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 references[1] 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

$$
\begin{aligned}
\vdash \Psi \;&\triangleq\; \forall \ell.\, \Psi(\ell) = \tau, \tau', h \Rightarrow \tau' \geqslant: \mathrm{head}(h) \\
\vdash \Sigma \;&\triangleq\; \forall \ell \in \mathit{dom}(\Sigma).\, (\Sigma(\ell) = (v, \mathsf{none}) \wedge v \notin \mathbb{L}) \\
&\qquad \vee (\Sigma(\ell) = (\ell', \mathsf{some}(\tau', \tau)) \wedge \exists v.\, v = \mathsf{pointsto}(\Sigma, \ell) \wedge \neg(v \sim {*}{\times}{*}) \vee \wedge v \sim \tau' \wedge v \sim \tau) \\
\Psi \vdash^{\ell} (v, \mathsf{none}) \;&\triangleq\; \exists \tau.\, \Psi(\ell) = [\tau] \\
\Psi \vdash^{\ell} (\ell', \mathsf{some}(\tau, \tau')) \;&\triangleq\; \Psi(\ell) = [\tau, \tau', \Psi(\ell')] \qquad\qquad h ::= \tau \mid \tau, \tau, h \\
\Psi \vdash \Sigma \;&\triangleq \vdash \Sigma \wedge \forall \ell \in \mathit{dom}(\Sigma) \cup \mathit{dom}(\Psi), \Psi \vdash^{\ell} \Sigma(\ell) \qquad\qquad \Psi : \ell \to h \\
\Sigma : (k, \Psi) \;&\triangleq \vdash \Psi \wedge \Psi \vdash \Sigma \wedge \forall \ell \in \mathit{dom}(\Sigma).\, (j, \Psi, \Sigma, \ell) \in \mathcal{VH}^{L}[\![\Psi(\ell)]\!]
\end{aligned}
$$

Fig. 10. Vigilance: Value-Log Type Satisfaction

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 Figure 9 show, we set up a Kripke logical relation[13] 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 Figure 10.

In more detail, as Figure 10 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 appended onto another typing history h', indicating type obligations added to the value by a cast expression. We say that a value log Σ satisfies a world $\Sigma : (k, \Psi)$ 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 value-log entry is well formed ($\vdash h$). 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 to the run-time typing history of a value, this gives rise to a syntactic constraint that the "from" type of such an entry matches (up to subsumption) the type that the casted value previously held in the history.

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 $v \sim_{L}^{cast} \tau$ checks performed by the cast evaluation rules) and are never pairs because our pairs are evaluated eagerly. For the latter, $\Psi \vdash^{\ell} (v, -)$ specifies that if the entry does not record a cast around a value, it is consistent with Ψ when its run-time typing history $\Psi(\ell)$ does not include any type obligations added by a cast. If the entry does record a cast, then the recorded types must match those in $\Psi(\ell)$, and the casted location must itself be well formed with respect to the remaining entries in the run-time typing history.

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 $\Sigma(\ell) \in \mathcal{V}^{L}[\![\tau]\!]$ for each $\tau \in \Psi(\ell)$. Since casts in our

$$
\begin{aligned}
\mathcal{VH}^L[\![C, \tau_2, \ldots \tau_n]\!] &\triangleq \{(k, \Psi, \Sigma, \ell) \mid \forall \tau \in [C, \tau_2, \ldots \tau_n].\ (k, \Psi, \Sigma, \ell) \in \mathcal{V}^L[\![\tau]\!]\} \\
\mathcal{VH}^L[\![\tau_1' \times \tau_1'', \tau_2, \ldots \tau_n]\!] &\triangleq \{(k, \Psi, \Sigma, \ell) \mid \Sigma(\ell) = (\langle \ell_1, \ell_2 \rangle, \_) \wedge\ (k, \Psi, \Sigma, \ell_1) \in \mathcal{VH}^L[\![\tau_1', \mathit{fst}(\tau_2), \ldots \mathit{fst}(\tau_n)]\!] \\
&\qquad \wedge\ (k, \Psi, \Sigma, \ell_2) \in \mathcal{VH}^L[\![\tau_1'', \mathit{snd}(\tau_2), \ldots \mathit{snd}(\tau_n)]\!]\} \\
\mathcal{VH}^L[\![\tau_1' \to \tau_1'', \tau_2, \ldots \tau_n]\!] &\triangleq \{(k, \Psi, \Sigma, \ell) \mid \forall (j, \Psi') \sqsupseteq (k, \Psi),\ \Sigma' \supseteq \Sigma,\ \ell_v,\ \tau_0 \geqslant: \tau_2.\ (j, \Psi', \Sigma', \ell_v) \in\ \mathcal{V}^L[\![\tau_1']\!] \\
&\qquad \wedge \Sigma' : (j, \Psi') \Rightarrow (j, \Psi', \Sigma', \mathsf{app}\{\tau_0\}\, \ell\ \ell_v) \in \mathcal{EH}^L[\![\, [\tau_0, \mathit{cod}(\tau_2), \ldots \mathit{cod}(\tau_n)]\,]\!]\} \\
\mathcal{VH}^L[\![*, \tau_2, \ldots \tau_n]\!] &\triangleq \{(k, \Psi, \Sigma, \ell) \mid (k-1, \Psi, \Sigma, \ell) \in \mathcal{VH}^L[\![C, \tau_2, \ldots \tau_n]\!] \cup \\
&\qquad \mathcal{VH}^L[\![* \times *, \tau_2, \ldots, \tau_n]\!] \cup \mathcal{VH}^L[\![* \to *, \tau_2, \ldots, \tau_n]\!]\}
\end{aligned}
$$

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

Fig. 11. Vigilance: Typing History Relations

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 $\mathcal{V}^L[\![\tau_1 \to \tau_2]\!]$ 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 $v \in \mathcal{V}^L[\![\tau_1]\!]$. 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 cast $\{\tau_1 \Leftarrow \mathit{dom}(\tau')\}\ v$[2] 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[3]. These relations, $\mathcal{VH}^L[\![\overline{\tau}]\!]$ and $\mathcal{EH}^L[\![\overline{\tau}]\!]$ are given in Figure 11. For a nonempty list of types $\tau, \overline{\tau}$, the relation is defined inductively over the first type in the list, following a similar structure to $\mathcal{V}^L[\![\tau]\!]$. When τ is a base type B, the value typing-history relation $\mathcal{VH}^L[\![B, \overline{\tau}]\!]$ contains any ℓ such that $\mathsf{pointsto}(\Sigma, \ell)$ is in $\mathcal{V}^L[\![\tau]\!]$ for each $\tau \in [B, \overline{\tau}]$. Just as in the value relation, because casts on pairs are evaluated eagerly, $\mathcal{VH}^L[\![\tau_1 \times \tau_2, \overline{\tau}]\!]$ contains only literal pairs $\langle \ell_1, \ell_2 \rangle$ whose components inductively satisfy all the appropriate types.

As discussed above, $\mathcal{VH}^L[\![\tau_1 \to \tau_2, \overline{\tau}]\!]$ requires a function to behave well only when it is given an argument $v \in \mathcal{V}^L[\![\tau_1]\!]$. As in the $\mathcal{V}$ relation, it must also behave well when the application is annotated with any $\tau_0 \geqslant: \tau_2$. Behaving "well" means that an application, evaluated with a future store $\Sigma' : (j, \Psi') \sqsupseteq (k, \Psi)$ should behave like all the types $\tau_0, \mathit{cod}(\overline{\tau})$. Since this is an expression, we define the $\mathcal{EH}^L[\![\overline{\tau}]\!]$ 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 $\mathcal{VH}^L[\![\overline{\tau}]\!]$ rather than $\mathcal{V}^L[\![\tau]\!]$.

Finally, as in the value relation, $\mathcal{VH}^L[\![*, \overline{\tau}]\!]$ is an untagged union over base types $\mathcal{VH}^L[\![B, \overline{\tau}]\!]$, pairs of dynamic types $\mathcal{VH}^L[\![* \times *, \overline{\tau}]\!]$, and functions between dynamic types $\mathcal{VH}^L[\![* \to *, \overline{\tau}]\!]$, at step index $k - 1$.

Vigilance: Top-level Relation. We now define vigilance for open terms in the standard way, see Figure 12. We extend the characterization of vigilance for closed terms and types to open terms

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

$$\mathcal{G}^L[\![\Gamma]\!] \triangleq \{(k,\Psi,\Sigma,\gamma) \mid dom(\Gamma) = dom(\gamma) \wedge \forall x(k,\Psi,\Sigma,\gamma(x)) \in \mathcal{V}^L[\![\Gamma(x)]\!]\}$$
$$[\![\Gamma \vdash_t e : \tau]\!]^L \triangleq \forall(k,\Psi,\Sigma,\gamma) \in \mathcal{G}^L[\![\Gamma]\!].\ \Sigma : (k,\Psi).\ \Rightarrow (k,\Psi,\Sigma,\gamma(e)) \in \mathcal{E}^L[\![\tau]\!]$$

Fig. 12. Vigilance: Top-Level Relation

$$\begin{array}{rcl} K & ::= & \mathsf{Nat} \mid \mathsf{Int} \mid \mathsf{Bool} \mid *\times* \mid *\rightarrow* \mid * \\ \Gamma & ::= & \cdot \mid \Gamma, (x{:}K_0) \\ e & ::= & x \mid n \mid i \mid \mathsf{True} \mid \mathsf{False} \mid \lambda(x{:}K).\,e \mid \langle e,e\rangle \mid \mathsf{app}\{K\}\,e\,e \\ & \mid & \mathsf{fst}\{K\}\,e \mid \mathsf{snd}\{K\}\,e \mid binop\,e\,e \\ & \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{cast}\,\{K \Leftarrow K\}\ e \end{array}$$

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

$$\textsc{T-App}\quad \frac{\Gamma_0 \vdash_{\mathsf{tag}} e_0 : *\rightarrow* \qquad \Gamma_0 \vdash_{\mathsf{tag}} e_1 : K_0}{\Gamma_0 \vdash_{\mathsf{tag}} \mathsf{app}\{K_1\}\,e_0\,e_1 : K_1}$$

Fig. 13. A tag type system for λ^{IGTL}

by defining $[\![\Gamma \vdash_t e : \tau]\!]^L$ which says that an expression e that type checks in context Γ behaves like τ when, given a substitution γ that behaves like Γ, $\gamma(e)$ behaves like τ. And 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 $\Gamma(x)$. We parameterize the typing judgment with type system t which ranges (so far) over sim for simple typing, and tag for tag typing.

With our vigilance logical relation in place, we formally define vigilance as the fundamental property of the vigilance relation.

Theorem 4.1 (Vigilance Fundamental Property). *If* $\Gamma \vdash_t e : \tau$ *then* $[\![\Gamma \vdash_t e : \tau]\!]^L$.

Vigilance has two components: values satisfy the types ascribed to them by the type system, and the types of all casts that were evaluated to produce them. The theorem says that values satisfy the types ascribed to them by the type system when we ask that well typed terms $\vdash e : \tau$ are in the expression relation $\mathcal{E}^L[\![\tau]\!]$, which says that if e runs to a value, then that value must behave like τ. Moreover, the theorem states values satisfy the types of all casts that were evaluated to produce them in the requirement that when e runs to a value with value log Σ', we also ask that there is a future world $(k-j,\Psi') \sqsupseteq (k,\Psi)$, where $\Sigma' : (k-j,\Psi')$.

Natural is vigilant for simple typing. Since Natural employs proxies to enforce the types of every cast it encounters during the evaluation of a term, we can show that it is vigilant for simple typing.

Theorem 4.2 (Natural is Vigilant for Simple Typing). *If* $\Gamma \vdash_{\mathsf{sim}} e : \tau$ *then* $[\![\Gamma \vdash_{\mathsf{sim}} e : \tau]\!]^N$.

5 THE TRUTH ABOUT TRANSIENT

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 IGTL program. For instance, consider $f = \mathsf{cast}\,\{\mathsf{Int} \rightarrow \mathsf{Int} \Leftarrow *\}\ \mathsf{cast}\,\{* \Leftarrow \mathsf{Nat} \rightarrow \mathsf{Nat}\}\ \mathsf{cast}\,\{\mathsf{Nat} \rightarrow \mathsf{Nat} \Leftarrow * \rightarrow *\}\ (\lambda x : *.\,x)$ and $e = \mathsf{app}\{\mathsf{Int}\}\,f\,{-1}$. Clearly, $\cdot \vdash_{\mathsf{sim}} e : \mathsf{Int}$ and $e \hookrightarrow_T -1$. However, the final result of e is also the result of f which is supposed to be a Nat according to f's casts. But, Transient does not perform the checks these casts entail. Hence, Transient is not vigilant for simple typing:

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

However, as previous work hints at [6, 7, 21], Transient does enforce the tags of all the types it encounters during the evaluation of an IGTL program. To formalize the relation between Transientand the enforcement of tags as a vigilance property, we first define a so-called tag type system for λ^{IGTL} (Figure 13). The rules of a tag type system relate a term with a tag K that corresponds to

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

τ ::= Nat | Int | Bool | $\tau{\times}\tau$ | $*{\longrightarrow}\tau$ | $*$ | $\bot$

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

$$\begin{tikzcd}[arrows=dash]
 & & * \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] & \\
\mathsf{Int} \arrow[d] & \mathsf{Bool} \arrow[ddr] & \tau_1'{\times}\tau_2' \arrow[d] & *{\longrightarrow}\tau_1' \arrow[d] \\
\mathsf{Nat} \arrow[drr] & & \tau_1{\times}\tau_2 \arrow[d] & *{\longrightarrow}\tau_1 \arrow[dl] \\
 & & \bot &
\end{tikzcd}$$

$$\frac{\Gamma \vdash_{\mathsf{tru}} e_1 : \bot \quad \Gamma \vdash_{\mathsf{tru}} e_2 : \tau_2}{\Gamma \vdash_{\mathsf{tru}} \mathsf{app}\{K\}\, e_1\, e_2 : \bot} \qquad \frac{\Gamma \vdash_{\mathsf{tru}} e_b : \mathsf{Bool} \quad \Gamma \vdash_{\mathsf{tru}} e_1 : \tau_1 \quad \Gamma \vdash_{\mathsf{tru}} e_2 : \tau_2}{\Gamma \vdash_{\mathsf{tru}} \mathsf{if}\ e_b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau_1 \sqcup \tau_2} \qquad \frac{\Gamma \vdash_{\mathsf{tru}} e : \tau}{\Gamma \vdash_{\mathsf{tru}} \mathsf{cast}\,\{K_2 \Leftarrow K_1\}\ e_0 : K_2 \sqcap K_1 \sqcap \tau}$$

Fig. 14. The Truer Type System

the top-level type constructor of the type. The typing rules for the tag type system are entirely unsurprising; the most interesting one is that for applications, which, as Figure 13 shows, allows arguments at any type, since the type of a function is simply the procedure tag that does not contain any information about the function's domain. Another important restriction of this setting is that the types of casts and assertions are also restricted to tags K. Since this is exactly the precision of types that Transient can enforce, the type ascriptions of an IGTL program should reflect that.

Overall, we can obtain a λ^{GTL} and λ^{IGTL} with tag typing, and a type-preserving translation between the two by lifting the meta function $\lfloor\cdot\rfloor$, which given a type τ returns the corresponding tag K, to the syntax definitions, the translation, and the typing rules from §3. As a final note, on tag typing, every program that simple type checks also tag type checks:

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

With tag typing in hand, we can show that Transient is indeed vigilant for it:[4]

Theorem 5.3 (Transient is Vigilant for Tag Typing). *If* $\Gamma \vdash_{\mathsf{tag}} e : K$ *then* $[\![\Gamma \vdash_{\mathsf{tag}} e : K]\!]^{\mathcal{T}}$.

Despite Theorem 5.2, tag types are not "weaker" than simple types: a function with type $*\longrightarrow*$ can be used safely in more contexts than a function with type $\tau\longrightarrow*$. Because of this difference tag typing is unsound for Natural. For instance, the expression $e = \mathsf{app}\{*\}\ (\lambda(x : *{\times}*).\, \mathsf{fst}\{*\}\, x)\ \mathsf{cast}\,\{* \Leftarrow \mathsf{Nat}\}\ 42$ has tag type $*$ but $e \hookrightarrow_N \mathsf{Wrong}$. 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. Consequently, Natural is not vigilant for tag typing:

Theorem 5.4 (Natural is Not Vigilant for Tag Typing).
There are Γ, e, K *such that* $\Gamma \vdash_{\mathsf{tag}} e : K$ *and* $\neg[\![\Gamma \vdash_{\mathsf{tag}} e : K]\!]^{\mathcal{N}}$.

5.1 A Truer Type System for λ^{IGTL}

While each individual Transient cast checks only a tag, because Transient is vigilant for tag typing extra information about a value is available to developers. For example, consider the function $f = \lambda(x : *{\times}*).\, x$. Under the tag type system, f type checks at $*\longrightarrow*$. From this type, a developer can deduce only that f is well-behaved when given any argument, and that it makes no promises about its result. However, vigilance implies that Transient also checks that the argument of f is a pair before returning it. Consequently, a developer should be able to conclude that the return type of f is really $*{\times}*$. In general, each time Transient performs a check, a developer can deduce that

[4]The relational interpretation of tag types is slightly different from that of simple types; it is modified in the same ways as the relation discussed in §5.1.

the checked value has both the tag that the check confirms, and the tag that the tag type system deduces for the checked expression.

To make this extra type information manifest statically, we design a *truer* type system for λ^{IGTL}. In the remainder of this section, we describe this type system, show that Transient is vigilant for it, and demonstrate how it enables the provably correct elision of some of the tag checks that Transient performs.

Figure 14 presents the truer type system. Just as for tag typing, its rules assume a restricted λ^{IGTL} 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 Transient function types is always $*$. After all, in Transient, the internal checks 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 Figure 14.

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 transient 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 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.

This type system accepts just as many programs as the tag type system, but calculates more precise types for them:

Theorem 5.5 (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 5.6 (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* $\tau' \leq \lfloor\tau\rfloor$.

The typing rule for conditionals is revealing of the limitations of truer typing. Even though the truer type system aims to reflect statically as much information as possible from the tag checks performed during the evaluation of a λ^{IGTL} expression, as all type systems, it fails to do so accurately. As a result, vigilance for truer typing is a stronger property than type soundness for truer typing. 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.

Transient *is vigilant for truer typing, but* Natural *is not.* The specifics of the truer type system require a few changes to the vigilance logical relation from §4. These changes are akin to the necessary changes to the logical relation for type soundness when one modifies a type system. Namely, we obtain a relational interpretation of truer types by (1) removing the restriction that an

$$\frac{\Gamma,\ (x{:}K) \vdash_{\mathsf{tru}} t \Leftarrow^{+} \tau \rightsquigarrow e}{\Gamma \vdash_{\mathsf{tru}} \lambda(x{:}K) \to \tau.\, t \Rightarrow * \to \tau \rightsquigarrow \lambda(x{:}K).\, e} \qquad \frac{\Gamma \vdash_{\mathsf{tru}} t \Leftarrow \tau \rightsquigarrow e}{\Gamma \vdash_{\mathsf{tru}} t \Leftarrow^{+} \tau \rightsquigarrow e} \qquad \frac{\begin{array}{c}\neg(\exists e.\, \Gamma \vdash_{\mathsf{tru}} t \Leftarrow \tau \rightsquigarrow e) \\ \Gamma \vdash_{\mathsf{tru}} t \Leftarrow^{\Rightarrow} \tau \rightsquigarrow e\end{array}}{\Gamma \vdash_{\mathsf{tru}} t \Leftarrow^{+} \tau \rightsquigarrow e}$$

$$\frac{\Gamma \vdash_{\mathsf{tru}} t \Leftarrow^{+} (\tau \setminus \lfloor \tau \rfloor) \times * \rightsquigarrow e}{\Gamma \vdash_{\mathsf{tru}} \mathsf{fst}\, t \Leftarrow \tau \rightsquigarrow \mathsf{fst}\{\lfloor \tau \rfloor\}\, e} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\mathsf{tru}} t \Rightarrow \tau' \rightsquigarrow e \\ \tau' \leq \tau\end{array}}{\Gamma \vdash_{\mathsf{tru}} t \Leftarrow^{\Rightarrow} \tau \rightsquigarrow e} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\mathsf{tru}} t \Rightarrow \tau' \rightsquigarrow e \\ \tau' \not\leq K\end{array}}{\Gamma \vdash_{\mathsf{tru}} t \Leftarrow^{\Rightarrow} K \rightsquigarrow \mathsf{cast}\ \{K \Leftarrow \lfloor \tau' \rfloor\}\ e}$$

Fig. 15. Truer Translation from λ^{GTL} to λ^{IGTL}

application annotation has to be a super type of the expected codomain type in the function case of the $\mathcal{V}$ and $\mathcal{VH}$ — unlike simple typing, truer typing accepts the type ascription of an application as it is anticipating that the semantics of the IGTL checks it (see Figure 14); (2) removing the constraint $\vdash \Psi$ from $\Sigma : (k, \Psi)$ — unlike simple typing, truer typing does not impose the constraint that an expression can be cast only from its deduced type to a compatible type, since the semantics is expected to check that a casted value matches the tag of both the "source" and "destination" type ascription of a cast; and (3) adding trivial cases for $\bot$ to the $\mathcal{V}$ and $\mathcal{VH}$ relations. (The full relation is shown in the supplemental material.) We can now use this logical relation for truer types to show that Transient is vigilant for truer typing:

Theorem 5.7 (Transient is Vigilant for Truer Typing). *If* $\Gamma \vdash_{\mathsf{tru}} e : \tau$ *then* $[\![\Gamma \vdash_{\mathsf{tru}} e : \tau]\!]^{T}$.

However, for the same reasons that Natural is not vigilant for the tag typing, neither is it vigilant for truer typing:

Theorem 5.8 (Natural is Not Vigilant for Truer Typing). *There are* Γ, e, τ *such that* $\Gamma \vdash_{\mathsf{tru}} e : \tau$ *and* $\neg[\![\Gamma \vdash_{\mathsf{tru}} e : \tau]\!]^{N}$.

5.2 From λ^{IGTL} with Truer Typing to λ^{GTL}

Adjusting λ^{GTL} to truer typing is not as straightforward as for tag typing. Recall that for tag typing, we obtain a restricted syntax for λ^{GTL}, a type system and a type-preserving translation simply by using the lifted meta function $\lfloor \cdot \rfloor$ on the corresponding definitions from §3. However, a λ^{GTL} with truer typing calls for a bidirectional type system and translation to λ^{IGTL} in order to capitalize on truer typing's ability to take advantage of the tag checks from casts to refine the types of expressions.

Figure 15 presents a few salient rules of the judgments that together define the type checker for λ^{GTL} expressions and their translation to λ^{IGTL}. The "infers" judgment of the bidirectional translation ($\Gamma \vdash_{\mathsf{tru}} t \Rightarrow \tau \rightsquigarrow e'$) is similar to that of the translation from §3; given a type environment and a λ^{GTL} expression, it type checks t at type τ and translates it to the λ^{IGTL} expression e. 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 $\Gamma \vdash_{\mathsf{tru}} t \Leftarrow^{+} \tau \rightsquigarrow e$ that attempts to construct a translated function body e' that has type τ. After all, the truer type system is supposed to reflect the types that Transient performs, and Transient only uses casts that perform tag checks and do not enforce a type τ otherwise.

The "checks" judgment itself employs two other judgments. $\Gamma \vdash_{\mathsf{tru}} t \Leftarrow \tau \rightsquigarrow e$ 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 Figure 17 for the definition of $\cdot \setminus \cdot$). $\Gamma \vdash_{\mathsf{tru}} t \Leftarrow^{\Rightarrow} \tau \rightsquigarrow e$ applies to any expression where the previous judgment does not apply. It attempts to recursively use the "infers" judgment 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 merely a tag).

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

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

The "infers" judgments translates the body of the function , but doesn't simply try to insert a single cast for the to enforce Nat×Nat, like the simple translation judgment from §3 would. Instead, it delegates to the "checks" judgment which attempts to find a way to insert type ascriptions 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 ascribes the pair projections in the body of the function with Nat, the tag of the required type, leading to the following translated λ^{IGTL} expression:

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

Importantly, however, just like the translation from §3, this translation is type-preserving:

Theorem 5.9 (The Translation From λ^{GTL} to λ^{IGTL} Preserves Truer Types). *If* $\Gamma \vdash t \Rightarrow \tau \rightsquigarrow e$ *then* $\Gamma \vdash_{\mathsf{tru}} e : \tau'$ *where* $\tau' \leq \tau$.

The fact that τ' may be below τ in the subtyping lattice implies that the evaluation of e in Transient must at least enforce the types of t, but might possibly also enforce even "stronger" types.

5.3 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. [20] use a sophisticated constraint system to infer when Transient's tag checks may be elided due to static information that the type system computes. Although the primary purpose of our truer type system is to provide the programmer with a faithful reflection of the static reasoning enabled by Transient's runtime checks, the static information it provides may very naturally be used to implement and prove correct a similar elision pass for Transient tag checks.[5]

For example, consider the variant of the example from §1 in Figure 16. Here, the developer creates two different type adapters for the library function `segment`. The truer types of `segment_png_small` and `segment_png` are different. Since the calls to `png_crop`'s ensure a tag check on each returned value, the first is certain to produce PNGs, while the second is not.

```
def segment_png_small(img : PNG) -> (PNG, PNG):
  let (a, b) = segment(img)
  in png_crop(a), png_crop(b)
def segment_png(img : PNG) -> (Dyn, Dyn):
  return segment(img)
png1: PNG = segment_png(...)[0]
png2: PNG = segment_png_small(...)[0]
```

Fig. 16. Two Type Adapters for an Image Library

When the developer attempts to project from the results of each of these functions, Transient checks in both cases that the result in 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 a check to be elided in one case but not the other, is reflected in their truer types!

In terms of the rules of the truer type system from Figure 14, all rules that involve an expressions 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

[5]Most notably, since our type system is not a whole-program (or whole-module) analysis, it assumes that any lambda may eventually be cast to $*$ and thereby confronted with arguments about which nothing is known statically; consequently, it does not seek to optimise the checks a function performs on its arguments.

$\boxed{\Gamma \vdash e : \tau \rightsquigarrow e}$ (selected rules)

$$\frac{\Gamma_0 \vdash e_0 : * \rightarrow \tau_1 \rightsquigarrow e_0' \qquad \Gamma_0 \vdash e_1 : \tau_0' \rightsquigarrow e_1'}{\Gamma_0 \vdash \mathsf{app}\{K_1\}\, e_0\, e_1 : K_1 \sqcap \tau_1 \rightsquigarrow \mathsf{app}\{K_1 \setminus \tau_1\}\, e_0'\, e_1'} \qquad \frac{\Gamma_0 \vdash e_0 : \bot \rightsquigarrow e_0' \qquad \Gamma_0 \vdash e_1 : \tau_0' \rightsquigarrow e_1'}{\Gamma_0 \vdash \mathsf{app}\{K_1\}\, e_0\, e_1 : \bot \rightsquigarrow \mathsf{app}\{K_1 \setminus \bot\}\, e_0'\, e_1'}$$

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

Fig. 17. Check-elision optimisation

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 ($\tau \not\leq K$).

Figure 17 provides an overview of a elision pass for redundant tag checks. The judgment $\Gamma \vdash e : \tau \rightsquigarrow e$ consumes a typing environment and a λ^{IGTL} expression e, type checks e at τ the same way as the truer type system for λ^{IGTL}, 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 corresponds the tag check that is necessary to enforce K given that τ is already known — in the subtyping lattice, this is $*$ if $\tau \leqslant: K$, and K otherwise. The elision pass preserves contextual equivalence:

Theorem 5.10 (Check-elision soundness). *If* $\Gamma \vdash e : \tau \rightsquigarrow e'$, *then* $\Gamma \vdash e \approx^{\mathrm{ctx}} e' : \tau$.

6 REFLECTIONS ON VIGILANCE AND LOOKING FORWARD

There are two possible directions for future work. The first concerns the improvement and evaluation of the truer type system. For instance, equipping the type system with occurrence types [18] or other path- and flow-sensitive features can help improve its precision. At the same time, implementing truer typing for a production language, and evaluating its performance and ability to detect issues in real codebases will provide guidance for how to grow the formal kernel of this paper to a useful tool for developers. The second future direction is theoretical and involves developing a framework for reasoning about contextual equivalence, such as an equational theory, based on vigilance.

Vigilance is the first semantic property that describes a gradual type system as two components, a static and a dynamic, that work together to help developers organize their code in a correct manner. When the static component relies on the dynamic, but the latter does not deliver, the meaning of types becomes misleading, and hence, developers may discover that their type-based assumptions about their code are false. When the dynamic component offers more than what the static component can capture, there is a missed opportunity for language designers. Vigilance is a compass for exploring the design space and finding a balance between the static and dynamic components of gradual typing. For instance, in this paper we use vigilance to show the combination of Transient and simple typing is not balanced, and neither is the combination of Transient and tag typing. In response, with vigilance as a guide, we propose a balance point for Transient. Specifically, we transfer some of the reasoning power due to the dynamics of Transient to the statics of our truer type system. We hope that vigilance can more generally be a tool for identifying opportunities to further incorporate reasoning patterns that developers routinely use in dynamically-typed languages into the design of gradual type systems [5, 18].

REFERENCES

[1] Amal Jamil Ahmed. 2004. *Semantics of Types for Mutable State.* Ph. D. Dissertation. USA.

[2] 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.

[3] Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *ACM SIGPLAN Symposium on Principles of Programming Languages.* 429–442.

[4] 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

[5] Michael Greenberg. 2019. The Dynamic Practice and Static Theory of Gradual Typing. In *3rd Summit on Advances in Programming Languages (SNAPL 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 6:1–6:20. https://doi.org/10.4230/LIPIcs.SNAPL.2019.6

[6] Ben Greenman and Matthias Felleisen. 2018. A Spectrum of Type Soundness and Performance. 2, ICFP (2018), 71:1–71:32. https://doi.org/10.1145/3234594

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

[8] Joshua Hoeflich, Robert Bruce Findler, and Manuel Serrano. 2022. Highly Illogical, Kirk: Spotting Type Mismatches in the Large Despite Broken Contracts, Unsound Types, and Too Many Linters. OOPSLA (2022), 142:1 — 142:26. https://doi.org/10.1145/3563305

[9] Erik Krogh Kristensen and Anders Møller. 2017. Type Test Scripts for TypeScript Testing. OOPSLA (2017), 90:1—90:25. https://doi.org/10.1145/3133914

[10] 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

[11] 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.

[12] 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

[13] 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

[14] 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.

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

[16] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: from Scripts to Programs. In *Dynamic Languages Symposium.* 964–974.

[17] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *ACM SIGPLAN Symposium on Principles of Programming Languages.* 395–406.

[18] Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical Types for Untyped Languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/1863543.1863561

[19] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages* (Portland, Oregon, USA) *(DLS '14)*. Association for Computing Machinery, New York, NY, USA, 45–56. https://doi.org/10.1145/2661088.2661101

[20] 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

[21] 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

[22] Philip Wadler and Robert Bruce Findler. 2009. Well-typed Programs Can't be Blamed. In *European Symposium on Programming.* 1–15.

[23] Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. 2017. Mixed Messages: Measuring Conformance and Non-Interference in TypeScript. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. 28:1—28:29.