Gradually Typed Languages Should Be Vigilant!

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

In gradual typing, different languages use different dynamic type checks even though they have the same static type system. This raises the question of whether the dynamic semantics of a gradually typed language enforces sufficiently the types of its static type system. Neither type soundness, nor complete monitoring, nor any other meta-theoretic property of gradually typed languages provides a satisfying answer.

In response, we present *vigilance*, a semantic analytical instrument that detects when the dynamic semantics of a gradually typed language is adequate for its static type system. Technically, vigilance asks if a dynamic semantics enforces the complete *run-time typing history* of a value, which consists of all the types associated with the value at run time. We show that Natural is *vigilant* for the standard simple type system, but Transient is not, while Transient is *vigilant* for a tag type system, but Natural is not, hence, clarifying their comparative type-level reasoning power. Furthermore, using vigilance as a guide, we develop a new gradual type system for Transient, dubbed *truer*, that is stronger than tag typing and more faithfully reflects the type-level reasoning power guaranteed by Transient dynamic semantics.

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

1 VIGILANCE, A NEW ANALYTICAL INSTRUMENT FOR GRADUAL TYPING

Gradual typing exhibits an impressive variety. While industrial approaches tend to ignore types after static type checking, academic ones prioritize soundness and turn types into some form of checks. How these checks behave varies greatly from design to design.[1] As a notable point of contrast, languages that implement the Natural dynamic[2] semantics [15, 19, 21] convert types to contract-like proxies; while languages that implement the Transient semantics [25] simply inject so-called dynamic tag checks at strategic spots in a program. This variety has a pragmatic background: sound gradual typing often incurs prohibitive performance costs [12], thus different design points derive from different ways of resolving the tradeoff between performance and guarantees. What this design space lacks, though, is a toolbox for analyzing that tradeoff.

In response, this paper contributes *vigilance*, a new semantic analytical instrument for gradual typing. Vigilance captures an intuitive fact; when a language designer modifies the semantics of a language but leaves the static type system as is, parts of the types of a program may go unchecked, which can invalidate standard type-based reasoning principles. If that is the case, there is an alternative static type system that the modified language *actually enforces*, which can serve as the basis of type-based optimizations and refactorings. Vigilance reifies this adequate design point: a language semantics is vigilant for a static type system if the two work together to enforce all the types of a program.

Vigilance sends a pragmatic signal to language designers. When the semantics of a language is not vigilant for its static type system, the designer can consider a type system with less precise types either as an alternative to the original one, or as the basis of semantics-preserving optimizations

[1] See Greenman et al. [9] for a detailed overview of the landscape.
[2] For the rest of the paper, we use "semantics" to refer to "dynamic semantics."

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.

and IDE tools. Conversely, when the semantics is vigilant for a static type system, the language designer can use vigilance as a compass to find either a stronger type system or a more performant dynamics. En route to achieving these goals, vigilance offers language designers the instrument to determine whether the dynamics is (in)adequate for the statics.

Vigilance builds on prior research efforts to create such an instrument for gradual typing [9–11, 15, 20, 21, 25, 26] and improves on them. In particular, it is stronger that both syntactic type soundness and its semantic counterpart: it asks not only that a program value *behaves according to its type*, but also that it *behaves according to its run-time typing history*. Vigilance is also both more fine-grained and stronger than complete monitoring [5, 11]; it can positively or negatively characterize semantics other than Natural, and it entails that the semantics performs the meaningful checks at the right times.

Besides developing vigilance, in this paper we evaluate vigilance by analyzing side-by-side the two most-studied semantics from the literature — Natural and Transient — uncovering new facts about both:

- Reconfirming previous results, we prove that Natural semantics is vigilant for the standard simple type system while the Transient one is not. However, we also show that Transient is vigilant for a tag type system but Natural is not. Put differently, even though, from the perspective of prior syntactic analysis [9–11, 21] Natural is stronger that Transient, from the semantic perspective of vigilance this not the case; each corresponds to a distinct adequate design point.
- We use vigilance to perform a design exercise and find adequate design points for Transient besides tag typing. Specifically, we show that there exists at least one vigilant type system for Transient, which we dub *truer*, that is semantically stronger than the tag type system. Furthermore, to demonstrate the benefits of the exercise, we use the truer type system to justify and prove correct an optimization that elides unnecessary dynamic type checks similar to prior work.

Outline. The remainder of the paper is organized as follows. §2 describes the necessary background, the key ideas behind vigilance, and compares it in detail with prior work.§3 presents the formal linguistic framework of the paper and §4 builds on that to define vigilance formally and prove that Natural is vigilant for simple typing. §5 develops the tag and truer type systems, 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 describes related work not already covered in §2, and §7 discusses future directions and concludes.

2 MOTIVATION AND THE MAIN IDEAS BY EXAMPLE

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

2.1 Natural **and** Transient **Semantics in one Framework**

At first look Natural and Transient seem vastly different. Natural relies on proxies that are difficult [12] (though not impossible [14]) to implement in a performant manner, while Transient offers a lightweight alternative where tag checks are in-lined at the start of function bodies, and around function applications and elimination forms. These differences raise the problem of a fair comparison between the two semantics.

To overcome this obstacle, following [10], we construct a linguistic setting that minimizes the differences between the two semantics to just the essential bits. Specifically, we decouple the syntax and static type system of the *source language*, which we refer to as the Gradually Typed Language (*GTL*) and which is based on the gradually typed λ-calculus [19], from the meaning of its programs. A GTL program obtains meaning via a *type-preserving* translation to an *intermediate language* with casts and type assertions (*ICTL*).[3] However, the semantics of the ICTL is not fixed; its reduction rules are parameterized, and tweaks of the parameters result in different dynamic type checks. Hence, this uniform setting allows us to study how the meaning of a GTL program changes as we vary the semantics of the ICTL while keeping both the type of the program and its ICTL image the same.

Figure 1 gives a taste of our GTL as the starting point of the discussion of the Natural semantics. Specifically, the snippet defines `encode`, which expects as its second argument a function from strings to integers, and then applies it to `rolling_hash`, which has no type annotations. Even though the static checker for the standard gradual simple type system does not have at its disposal the necessary type information to derive that `rolling_hash` indeed has type 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 $*$[4] to the expected type 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.

```
let rolling_hash = λ el. (...)
let encode = λ file_path, (word_coder : String → Int). (...)
encode "/paper.tex" rolling_hash
```

Fig. 1. Dynamically Typed Argument for a Typed Parameter

The same code evaluates differently with Transient. As before, the translation injects a cast from type $*$ to type 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 `word_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.

The above discussion appears to rely on two different translations and two different evaluations languages — one with higher-order casts and one with first-order casts and type assertions. However, our framework employs a common translation from a GTL program to its ICTL image that has both casts and type assertions. The translated ICTL program can run either in a Natural or a Transient manner: in the first case the casts behave as higher-order proxies and the assertions are no-ops; in the second, casts and assertions perform immediate first-order checks. In other words, the difference between the two semantics boils down to ICTL dynamics.

2.2 The Gap Between Statics and Dynamics for Transient

While Transient does guarantee some type-level facts, in general these facts do not align deeply with standard type-based reasoning principles. This becomes clear when dynamically typed code uses a function with type other than $*$. In fact, Greenman et al. [11] describe a scenario with Transient where a library with a typed interface is in fact dynamically typed and the interface is

[3] ICTL plays the role of the cast calculus from the literature. Given that the term is overloaded and different variants have subtly different features, we introduce new nomenclature to avoid conflating terminology.

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

nothing more than a thin veneer of possibly misleading type annotations. In such cases, the casts and assertions injected by the translation of the GTL code to ICTL may not perform all the checks that are necessary to validate these types; it all depends on whether the dynamic semantics of the ICTL enforces the types of the interfaces. As a result, optimizations or refactorings that one might expect to hold if the types were enforced may not be valid.

Figure 2 adapts our running example to the scenario from Greenman et al. [11]. In this example, `encode` comes with no type annotations. Instead, `type_and_launder` acts a hand-rolled type adapter that aims to help the implementation of the dynamically typed `encode` take advantage of type information for its `word_coder` argument. In particular, one may expect that assigning the result of `type_and_launder` to `word_coder_laundered` makes unnecessary the use of defensive code that inspects the results of calls to `word_coder` to determine whether they are integers. However, whether calls to `word_coder_laundered` are checked to produce integers depends on the semantics of the ICTL. In Natural, `word_coder_laundered` has the proxy that inspects the results of the function, ensuring the type Int from the annotation; in Transient, there is no proxy and, since `word_coder` is an identifier of type $*$, the compiler injects no type assertions for the results of calls to `word_coder`, meaning the result is not guaranteed to be an integer.

```
let type_and_launder = λ (f : [String] → Int). f
let encode = λ file_path, word_coder.
   let word_coder_laundered =
      type_and_launder word_coder
   (...)
encode "/paper.tex" rolling_hash
```

Fig. 2. Applying a Hand-Rolled Type Adapter

2.3 Type Soundness is Not Enough

At first glance, this difference between the proxy enforcement in Natural and tag-based checks in Transient seems like an issue that type soundness should clarify. Type soundness does distinguish between the two [10], but falls short of fully characterizing this difference in their behavior. 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. Syntactic type soundness says that if a source program is well-typed at type τ and its ICTL image evaluates to a value, then, in Natural, the value also has type τ. In contrast, in Transient the ICTL image's result value has a type that matches the tag of τ (i.e. its top type constructor), but that is not necessarily exactly equal to τ (tag soundness). Hence, syntactic type soundness seems to reveal that the choice of ICTL semantics affects the predictive power of the GTL's type system.

However, syntactic type soundness stops short of explaining whether the ICTL semantics performs all the checks the GTL type system relies on. After all, syntactic type soundness only connects the type of the source program with the type of the translated ICTL program's result. Therefore, all intermediate types that do not contribute to that goal are immaterial. The running example in Figure 2 demonstrates this situation. First, given the pervasive lack of annotations in the example, the type system determines that the type of `word_coder_laundered` is $*$, which is inhabited by all values, not just functions from strings to integers. So type soundness says nothing about `word_coder_laundered` directly. Second, one cannot rely on the compositional nature of the typing rules of the type system and type soundness to deduce transitively some more precise type-level information than $*$ for `word_coder_laundered`. While the typing rules of our GTL, type soundness, and the details of the Natural translation and semantics do stipulate that the result of

`type_and_launder(word_coder)` needs to behave as a function from strings to integers, this is a fact that the type system cannot prove without the help of the semantics of the underlying ICTL, again due to the sparsity of type annotations.

As discussed above, in Natural, which employs a proxy around `type_and_launder(word_coder)`, the expected type is indeed enforced. But, in Transient, which does not employ proxies, the type, and the corresponding annotation, is "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 one cannot use type-level reasoning alone to justify the elision of defensive checks on the results of `word_coder_laundered`. In fact, the Forgetful [7] and Amnesic [11] variants of Natural create the same proxy as Natural but then remove it upon the assignment to `word_coder_laundered` 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 syntactically type sound as Natural. The root of the issue is that from the perspective of syntactic type soundness, it does not matter whether the result type of `type_and_launder(word_coder)` is ever enforced, meaning we must reason beyond syntactic type soundness to ensure types are meaningful.

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

2.4 Complete Monitoring is Not Enough

An attempt to deal with the issue of intermediate unchecked types is complete monitoring for gradual types [9, 11], which adapts the notion of complete monitoring from work on contract systems [5]. The starting point for complete monitoring is a collection of semantics for an ICTL, i.e., different semantics with the same syntax and the same simple type system. The goal is to determine which of the semantics enforces the types of ICTL programs completely. However, complete monitoring establishes a weaker property. Intuitively, an ICTL semantics is a complete monitor if it "has complete control over every typed-induced channel of communication between two components." [9].

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

Back to the running example in Figure 2, since Natural is a complete monitor, every "channel" is checked. Assuming that `word_coder` and `type_and_launder` have labels l_1 and l_2 respectively, as the first flows through the second, `word_coder` becomes `word_coder_laundered =` $(\texttt{cast \{[String] -> int\} word\_coder}_{l_1})_{l_2}$. While the term has multiple labels, they are separated by a cast , and hence, the semantics maintains the single-owner policy. Furthermore, when `word_coder_laundered` is applied in the body of `encode`, the casts perform checks that discharge

the accumulated ownership labels for the result of the function and allow it to obtain the same label as its calling component.

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

In summary, while complete monitoring offers a dimension that type soundness lacks, i.e., that casts should mediate the interactions of components, it is not strong enough to determine whether an ICTL semantics is adequate for a GTL type system. First, its formulation renders it coarse-grained as it cannot shed light on the guarantees Transient offers. Second, it requires defining a brittle syntactic system of annotations, along with an instrumented semantics to propagate them. Finally, even in this restricted setting, it entails a rather weak relation between the types of the ICTL and the checks that its semantics perform.

2.5 Enter Vigilance

Intuitively, a GTL should come with reliable type enforcement, so that types can be used for reasoning. Returning to the example in Figure 2, under either Natural or Transient, the result of `type_and_launder(word_coder)` should live up to the return type of `type_and_launder`, and similar for `word_coder_laundered` after the assignment.

To capture this intuition, this paper develops a new semantic property called *vigilance*. The goal of vigilance is to describe how the semantics of an ICTL impacts the enforcement of the types in a GTL program. Vigilance as a semantic property serves to identify when the dynamics of the ICTL are inadequate for the statics of a GTL, and goes beyond (semantic) type soundness and complete monitoring. By requiring that the ICTL semantics enforce every type obligation a value acquires during the evaluation of a program, **vigilance considers flow-sensitive and compositional type information that type soundness ignores**. And, instead of only specifying the location of checks as in complete monitoring, **vigilance ensures and allows for exactly enough dynamic enforcement to enforce the type-based properties of terms**. And by varying the type system and translation of a GTL, and thereby varying the typing histories for expressions, **vigilance is applicable to more systems than complete monitoring**.

2.6 Vigilance: Language Framework

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

For the first challenge, for each type system that we wish to consider, we define a single type-preserving translation that maps a well-typed GTL program of type τ to a well-typed ICTL program of the same type. Since the translation is type preserving, we reduce the relationship between ICTL semantics and GTL types to the relationship between ICTL semantics and ICTL types.

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

Unlike complete monitoring, the instrumented semantics in our framework are just a system for naming values and tracking types over a predefined semantics, which is mostly mechanical.

2.7 Vigilance: Technical Definition and Implications

After equipping our ICTL with two different instrumented semantics, for Natural and Transient respectively, we define vigilance using a (step-indexed) unary logical relation that models ICTL types τ as sets of values v that inhabit them. In contrast to a typical logical relation for type soundness [2], vigilance comes with an extra index, the *typing history* Ψ, which can be thought of as a value-log typing. Specifically, the typing history Ψ is a log that collects the types present on each cast or assertion in the program that a particular value has passed through.

While semantic type soundness says that a language is semantically type sound if and only if any well-typed expression $e : \tau$ "behaves like" τ, vigilance asks that e also "behaves" according to its typing history. We say that the semantics of an ICTL are *vigilant* for a type system if in any well-formed typing history Ψ (capturing potential casts and assertions from a context), the translation of e behaves like τ. The latter means that, if evaluating e produces a label ℓ (as well as a potentially larger typing history Ψ', capturing casts and assertions present in e), then $\Sigma(\ell)$ not only behaves like τ (in the conventional sense), but also like all of the types in $\Psi'(\ell)$.

2.8 Vigilance: By Example

A concrete discussion about vigilance requires an illustration of the contents of the typing history of a value. To that end, we analyze the evaluation of the example in Figure 3.

The example consists of three helper functions and a main one. Helper function `make_nat` coerces an integer into a natural number. Its purpose is to make sure that the arguments of helper function `use_nat` are naturals. The details of `use_nat` do not matter, but the important bit herein is that it raises an error when its argument is a negative integer. Helper function `use_int` is similar to `use_nat` except that it expects integer argument. The main function, `case_by_case`, takes an integer `i` and a boolean `b`, and performs two case analyses on `b`. The first conditional casts `i` to a natural when `b` is true. The result of the conditional is then assigned to variable `x`. The second conditional also inspects `b` and if it is true, it treats `x` as a natural and passes it to `use_nat`, otherwise it passes `x` to `use_int`. Different from previous examples and for presentation purposes only, the example comes with partial label annotations. Specifically, we write $(e)_l$ to denote an expression e annotated with a label l that uniquely identifies the value e evaluates to. Put differently, the labels are a layer of indirection so that we can refer to program values and relate them with types.

```
let make_nat = λ(i:Int) → Nat. i
let use_nat = λ(n:Nat). (...)
let use_int = λ(i:Int). (...)
let case_by_case = λ (i:Int), (b:Bool).
  let x = (if b then (make_nat i_l1)_l2 else i)_l0
  (...) # doesn't use x or b
  if b then use_nat x else use_int x
```

Fig. 3. Example to Illustrate Labels and Typing History

From the perspective of typing history, the second conditional is particularly interesting as it is a manifestation of type-based flow-sensitive reasoning. In more detail, if `make_nat` guarantees that its result is a natural then the calls `use_nat(x)` and `use_int(x)` are guaranteed to succeed. As a justification of this conclusion, consider the typing history of l_0, the label of the value of `x`. In the case that `b = False`, the first conditional of the example evaluates to `i`, so the typing history of l_0 is Int. Hence, in this case, in the second conditional `x` is safely passed to `use_int(x)`. In the case that `b = True`, the first conditional follows its first branch, and evaluation proceeds with the application of `make_nat`, which asserts that `i` must be a natural number. In the case that `i` is indeed a natural number, the cast succeeds, and l_0 inherits its typing history from l_2, which in turn

extends the typing history of l_1. As a result, the typing history of l_o is Nat and Int. Therefore, x is safely used as a natural number in this case too.

It is worth noting that the above discussion does not depend on what checks the dynamic semantics of the language performs, but only on the fact that the typing history of l_o is enforced. Consequently, any combination of statics and dynamics that enforces the typing history enables the described typed-based reasoning. Vigilance generalizes this point: when a semantics is vigilant for a semantics, during the evaluation of a program, every value satisfies all types in its typing history, and hence code that uses a value can safely assume that much. For instance, the example in Figure 2 shows that the simple type system and Natural work together to enforce the typing history of all values; the value bound to `word_coder_laundered` after the assignment behaves not only according to type $*$, but also, due to its type history, according to type String $\to$ Int. In contrast, the same is not true for the simple type system and Transient.

2.9 Vigilance: An Examination of Transient

Since vigilance brings to light details about the enforcement of types with Transient semantics, we revisit Transient and investigate: *for what type system is the* Transient *semantics vigilant?* An initial answer, which confirms prior work on Transient and tag soundness, is that one such type system ascribes type tags (top-level type constructors) to expressions rather than types. Hence, this *tag type system* accepts programs that have severely imprecise types, which makes plain the difference between what the simple type system promises and what the Transient semantics seems to achieve.

However, as the preceding section reveals, Transient has deeper type-level reasoning than tag soundness suggests. And the typing history that is a central piece of vigilance makes this additional power plain. As another example that demonstrates this point, consider the code in Figure 4. The example 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 previous examples, a typed interface for this library restricts it to PNGs. 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. Due to tag vigilance, the evaluation of the example in Transient is guaranteed to enforce the return type of `segment_png_small`: vigilance for tag typing entails that Transient checks that the results of the calls to `png_small` in the body of `segment_png_small` are PNGs. Hence, it is possible to deduce that result has type PNG×PNG, not just $*\times*$. In conclusion, Transient could be vigilant for a stronger type system, namely one which recognizes `segment_png_small` has return type PNG×PNG.

```
# Typed Interface
let png_crop = λ(img:PNG) → PNG. crop img
let segment_png_small = λ (img:PNG) → PNG×PNG.
   let (a, b) =
      segment img
   (png_crop a, png_crop b)
let foreground = (segment_png_small my_png)[0]
write_png foreground "fg.png"
```

Fig. 4. Using a Type Adapter for an Image Library

2.10 Vigilance: Towards Truer Transient Types

As a design exercise, we use vigilance as a guide towards a revised static type system that offers richer information than a tag type system, and define the *truer type system*. In detail, this type system makes limited use of union and intersection types in order to reflect the outcomes of Transient casts and assertions in the types of expressions.

The truer type system shows how vigilance can benefit language designers. They can start from an adequate design point and, with vigilance as a guide, find others. If a new adequate design point involves a more precise type system, such as the truer one, the designer may use it as part of IDE and refactoring tools, or for optimizing the dynamics of the language. For instance, a consequence of truer is that some of the checks that are necessary so that 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` indeed has a type that precisely matches its type annotation (rather than that it simply returns a pair that should be checked further at run time). Moreover, this precise truer type makes unnecessary a type assertion on the outcome of the left-pair projection that gets bound to `foreground`; it is statically known that it is a `PNG`. Hence, in §5.3 we use the truer type system to eliminate unnecessary transient checks in a semantic and vigilance preserving manner.

2.11 Technical Contributions

The following table summarizes the vigilance 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 programs expose that tag and truer typing are not strictly semantically weaker than the simple type system; they allows a function to be composed with more contexts than the simple type system. Transient's checks protect the function from these additional contexts while Natural's do not. Hence, vigilance characterizes Transient positively in a way that distinguishes it from Natural.

3 FROM A GTL TO AN ICTL WITH TWO SEMANTICS

The top portion of Figure 5 presents the syntax of λ^{GTL}, our GTL, which as described in §2 follows the approach of Greenman and Felleisen [10] and the gradually-typed λ-calculus [19]. Most of the features of λ^{GTL} are the same as the corresponding features of a simply-typed λ-calculus extended with constants, pairs and their relevant elimination forms. The one unconventional syntactic form is that for anonymous functions. In particular, anonymous functions come with type annotations that describe both the type of their arguments and the type of their result. The type annotations τ range over *simple types* with the addition of $*$, the dynamic type, which, as usual in the gradual typing setting, indicates imprecise or missing type information. For example, the expression $\lambda(x : * \to *) \to *.\, t$ represents an anonymous function that consumes functions and may return anything.

Since λ^{GTL} expressions t do not evaluate directly but are translated to an ICTL, before delving into the type checking and evaluation of λ^{GTL} expressions, we discuss briefly the syntax of λ^{ICTL}, our ICTL. The bottom portion of Figure 5 shows the syntax of λ^{ICTL} expressions e. Its features correspond to those of λ^{GTL} with a few important differences. First, functions $\lambda(x : \tau).\, e$ come with type annotations for their arguments but not their results. Second, pair projections and function applications also have type annotations. Third, λ^{ICTL} has a new syntactic form compared to λ^{GTL}:

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

$$
\mathit{binop} \quad ::= \quad \mathsf{sum} \mid \mathsf{quotient}
$$

$$
n \in \mathbb{N}\ , i \in \mathbb{Z}
$$

Fig. 5. Syntax of λ^{GTL} (top) and λ^{ICTL} (bottom)

cast expressions. Specifically, cast $\{\tau_1 \Leftarrow \tau_2\}\ e$ represents a cast from type τ_2 to τ_1 for the result of expression e. In other words, while in λ^{GTL} all type annotations are on functions, in λ^{ICTL}, they are spread over applications, pair projections, function parameters, and casts. This is because the first three are the syntactic loci in a program that correspond to "boundaries" between pieces of code that can have types with different precision according to the type system of the GTL. Hence, the translation injects type assertions and casts exactly at these spots.

Figure 6 presents type checking for λ^{GTL} expressions t and their translation to λ^{ICTL} 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 $\tau \to \tau'$ 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^{ICTL} function whose body is wrapped in a cast from τ'' to τ', if needed. Specifically, metafunction $[\tau \swarrow \tau']e$ inserts a cast around e when τ is *compatible* with τ' (written $\tau \sim \tau'$) but not a subtype of τ' (subtyping is standard). Compatibility is the reflexive and symmetric relation that rules out non-convertible type casts, or type casts that will always error. Unlike standard definitions, compatibility includes $\mathsf{Nat} \sim \mathsf{Int}$,[5] to allow programmers to freely convert between Naturals and Integers, and have the translation insert appropriate checks.

Conditionals type check and translate in a recursive manner. The type of the conditional is the consistent subtype join $\widetilde{\sqcup}$ of the types of its two branches. The consistent subtype join definition is standard [3], and gives the least upper bound of the types with respect to subtyping, as well as more precise types in place of $*$. To bridge the potential gap between the type of the branch and the consistent join, the translation may wrap each branch in a cast with the same metafunction as above. Translated applications obtain 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

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

and its λ^{ICTL} image

$$
\begin{array}{c}
e = \mathsf{app}\{*\}\ (\lambda f : \mathsf{Nat} \to \mathsf{Nat}.\, \mathsf{cast}\,\{* \Leftarrow \mathsf{Nat}\}\ (\mathsf{app}\{\mathsf{Nat}\}\ f\ 42)) \\
(\mathsf{cast}\,\{\mathsf{Nat} \to \mathsf{Nat} \Leftarrow * \to *\}\ \lambda x : *.\, x)
\end{array}
$$

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

[5] Our compatibility is a symmetric version of consistent subtyping [3] and if we used compatible subtyping here, we would not be able to convert from Int to Nat without going through a function with return type $*$.

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

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

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

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

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

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

$\boxed{\tau \sim \tau'}$ (Compatibility, selected rules)

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

Fig. 6. Simple Typing Translation From λ^{GTL} to λ^{ICTL}

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

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

Fig. 7. Simple Typing for λ^{ICTL}

translation annotates all applications with their expected types, affirming the return types from the type annotation in t.

Figure 7 gives some rules for the typing judgment for λ^{ICTL}: $\Gamma \vdash_{\mathsf{sim}} e : \tau$. In general, the type system of λ^{ICTL} is straightforward and closely follows the translation of λ^{GTL}.

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

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

ICTL Type Guarantees Lift to the GTL. As discussed in §2, the type-preserving translation allows us to focus on λ^{ICTL} to determine the impact of Natural and Transient on the enforcement of the types of a well typed λ^{GTL} expression t. Since, the semantics of an λ^{GTL} expression t of type τ is defined by elaboration into a λ^{ICTL} expression e also of type τ, the typing history of all values produced by the evaluation of t is enforced only if the the typing history of all values produced by the evaluation of e is enforced. Therefore, the question of whether a semantics for λ^{ICTL} is vigilant for its static type system reduces to whether the semantics of λ^{GTL} is vigilant for the latter's static type system.

Note: The complete formal development of λ^{GTL} and λ^{ICTL} 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^{ICTL}

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

To record these types, we devise a *log-based* reduction semantics for λ^{ICTL}. This 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 dynamic semantics maintains a *value log* Σ, which is a map from labels ℓ to values v and potential types $option(\tau \times \tau)$. The type information is optional because a value may never go through a cast.

The definition of the *log-based* reduction semantics requires an extension of the syntax of λ^{ICTL} with values, labels, unannotated applications, errors and, most importantly, expressions that correspond to the run-time representations of type casts and assertions. Essentially, these act as hooks that allow us to define either a Natural or a Transient semantics for λ^{ICTL} while leaving the rest of the formalism unchanged. The top left of Figure 8 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^{ICTL}: $\mathsf{Err}^{\bullet}$ are expected errors and include failures due to failed type casts and assertions, and Err° are unexpected errors that indicate a failure of type soundness such as a call to a value that is not a function. Err ranges over these.

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

The bottom part of Figure 8 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;

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

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

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

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

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

$$\Sigma, \mathsf{app}\{\tau_0\}\ \ell_0\ \ell_1 \hookrightarrow_L \Sigma, \mathsf{assert}\,\tau_0\ (\ell_0\ \ell_1)$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 8. Evaluation Syntax and Reduction Semantics for λ^{ICTL}

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^{ICTL} 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^{ICTL} relies on them. Specifically, a monitor expression performs any checks a proxy would perform using the compatibility metafunction, and produces a fresh label ℓ to record in the value log and associates ℓ with two additional types. Upon an application of a label, $\hookrightarrow_L$ retrieves the types associated with it, and creates a monitor expression to enforce them. Hence, if the compatibility metafunction does perform tag checks for monitor expressions, monitors implement the two steps of checking types with proxies: checking first-order properties of the monitored value, and creating further proxies

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

Fig. 9. Example of log-based reduction.

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

Fig. 10. Vigilance: Value and Expression Relations

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 9 gives a taste of the log-based semantics of λ^{ICTL} 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 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 VIGILANCE, FORMALLY

In this section, we define *vigilance*, a semantic property of an ICTL 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 10 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.

For base types, ℓ belongs to the relation $\mathcal{V}^L[\![B]\!]$ if $\text{pointsto}(\Sigma, \ell)$ 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.[6]

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[2] to (1) keep track of the run-time typing histories of values in a type history Ψ, just as models of references keep track of the types of references in a store typing, and (2) ensure that values in Σ satisfy the run-time typing histories in Ψ, just as models of references ensure that the store S satisfies the store typing.

Thus, as the highlighted parts in Figure 10 show, we set up a Kripke logical relation[2, 18] 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 11.

In more detail, as Figure 11 shows, our typing history Ψ associates with each label in the value log a run-time typing history h, where h is either a single type, indicating that the value was produced at that type, or h is two types τ, τ' appended onto another typing history h', indicating type obligations added by a cast expression that casts from τ' to τ. The head of a typing history

[6]Note that the logical relation is well founded despite the use of τ_0 here because our subtyping relation (see supplemental material) is such that τ_0 must be a type of the same size as τ_2. The only interesting subtyping rule is $\text{Nat} \leqslant: \text{Int}$; the rest capture reflexivity, transitivity, or recur on the structure of types.

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

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

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

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

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

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

Fig. 11. Vigilance: Value-Log Type Satisfaction

h is the top-most type τ when h is τ, or τ when h is τ, τ', h'. We say that a value log Σ satisfies a world, written $\Sigma : (k, \Psi)$, when three things are true:

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

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

3. The values in the log must satisfy their run-time typing history. The core semantic condition of value-log type satisfaction is that $\Sigma(\ell)$ must behave like each type τ in its run-time typing history $\Psi(\ell)$. But we cannot simply ask that $\Sigma(\ell) \in \mathcal{V}^{L}[\![\tau]\!]$ for each $\tau \in \Psi(\ell)$. Since casts in our model may be coercive, they can only be expected to function appropriately when the value passed to the cast is of the appropriate (semantic) type. Because $\mathcal{V}^{L}[\![\tau_1 \to \tau_2]\!]$ 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 $\{\tau_1 \Leftarrow dom(\tau')\}\, v$[7] 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[8]. These relations, $\mathcal{VH}^{L}[\![\overline{\tau}]\!]$ and $\mathcal{EH}^{L}[\![\overline{\tau}]\!]$ are given in Figure 12. For a nonempty list of types $\tau, \overline{\tau}$, the relation is defined inductively over the

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

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

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

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

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

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

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

Fig. 12. Vigilance: Typing History Relations

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

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

Fig. 13. Vigilance: Top-Level Relation

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 $\tau_0, cod(\overline{\tau})$. 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 13. We extend the characterization of vigilance for closed terms and types to open terms by defining $[\![\Gamma \vdash_{\mathsf{t}} e : \tau]\!]^L$ 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* $\Gamma \vdash_{\mathsf{t}} e : \tau$ *then* $[\![\Gamma \vdash_{\mathsf{t}} e : \tau]\!]^L$.

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

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

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

Fig. 14. Tag Typing for λ^{ICTL}

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 λ^{ICTL} Simple Typing). *If* $\Gamma \vdash_{\mathsf{sim}} e : \tau$ *then* $[\![\Gamma \vdash_{\mathsf{sim}} e : \tau]\!]^{\mathsf{N}}$.

As described in §3, the type-preserving translation for simple typing can be composed with the vigilance theorem to get a vigilance result for λ^{GTL} expressions.

Theorem 4.3 (Natural is Vigilant for λ^{GTL} Simple Typing).
If $\Gamma \vdash_{\mathsf{sim}} t : \tau \rightsquigarrow e$ *then* $[\![\Gamma \vdash_{\mathsf{sim}} e : \tau]\!]^{\mathsf{N}}$.

5 TRANSIENT IS MORE THAN TAG CHECKING

Transient does not use proxies and, as a result, it does not enforce all the types from casts it encounters during the evaluation of an ICTL program. For instance, consider $p = \langle -1, -1 \rangle$, $f = \lambda(x{:}\mathsf{Int} \times \mathsf{Int}) \to (\mathsf{Nat} \times \mathsf{Nat}).\, x$, and $t = f\ p$. The translation of t is

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

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

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

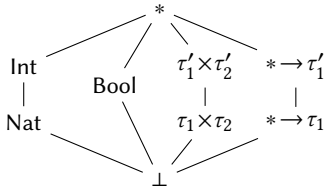
However, as previous work hints at [10, 11, 25], Transient does enforce the tags of all the types it encounters during the evaluation of an ICTL program. To formalize the relation between Transient and the enforcement of tags as a vigilance property, we first define a so-called tag type system for λ^{ICTL} (Figure 14). The rules of a tag type system relate a term with a tag K that corresponds to 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 14 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 ICTL program reflect that.

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

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

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

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



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

Fig. 15. Truer Typing for λ^{ICTL}

Overall, we can obtain a λ^{GTL} and λ^{ICTL} with tag typing, and a type-preserving translation between the two by lifting the "tag-of" metafunction $\lfloor\cdot\rfloor$, which given a type τ returns the corresponding tag K, to the syntax definitions, the translation, and the typing rules from §3.

$$\Gamma \vdash_{\mathsf{tag}} t : K \rightsquigarrow e \text{ iff } \exists \tau.\ \lfloor\tau\rfloor = K \wedge \Gamma \vdash_{\mathsf{sim}} t : \tau \rightsquigarrow e' \wedge e = \lfloor e' \rfloor$$

As a final note, 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:[9]

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

Despite Theorem 5.2, tag types are not semantically "weaker" than simple types: a function with type $* \to *$ can be used safely in more contexts than a function with type $\tau \to *$. Because of this difference tag typing is unsound for Natural. For instance, the GTL expression $(\lambda(f{:}*) \to *.\, f\ 42)\ (\lambda(x{:}*\times*) \to *.\, \mathsf{fst}\, x)$ translates to

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

which 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 λ^{GTL} Tag Typing).
There are Γ, t, K, e *such that* $\Gamma \vdash_{\mathsf{tag}} t : K \rightsquigarrow e$ *and* $\neg[\![\Gamma \vdash_{\mathsf{tag}} e : K]\!]^{\mathsf{N}}$.

5.1 A Truer Type System for λ^{ICTL}

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

[9]The relational interpretation of tag types is slightly different from that of simple types (see supplemental material); it is modified in the same ways as the relation for truer types which we will discuss in §5.1.

This combined with the fact that Transient does not satisfy vigilance for simple typing suggests that there is an alternative static type system that Transient actually enforces: we can capture *much more* static information with *no change* to the dynamics. As an exercise, we can make this extra static information manifest in a *truer* type system for λ^{ICTL}. In the remainder of this section, we describe this type system, 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 15 presents the truer type system. Just as for tag typing, its rules assume a restricted λ^{ICTL} syntax where type ascriptions are tag K. Similarly, type environments Γ map variables to tags. However, truer typing deduces more precise types τ than tags. These differ from simple types in two major ways. First, the domain of 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 15. Like other systems with subsumption rules for subtyping, the truer type system includes a subsumption rule, but for the subtyping lattice $\leq$.

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

Because applications perform a tag check on the result of the application, rather than typing the entire expression at the codomain of the function τ_1, the truer 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* $\Gamma \vdash_{\text{tag}} e : K$. *Then there exists some* $\tau \leq K$ *such that* $\Gamma \vdash_{\text{tru}} e : \tau$.

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

Corollary 5.6 (Simple Typing Implies Truer Transient Typing).
If $\Gamma \vdash_{\text{sim}} e : \tau$, *then* $\lfloor \Gamma \rfloor \vdash_{\text{tru}} \lfloor e \rfloor : \tau'$ *where* $\tau' \leq \lfloor \tau \rfloor$.

While the truer type system aims to reflect statically as much information as possible from the tag checks performed during the evaluation of a λ^{ICTL} expression, the static approximation in the conditional rule shows, as all type systems, it fails to do so accurately. As a result, 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.* 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 application annotation has to

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

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

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

Fig. 16. Truer Translation from λ^{GTL} to λ^{ICTL}

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 ICTL checks it (see Figure 15); (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 λ^{ICTL} Truer Typing). *If* $\Gamma \vdash_{\mathsf{tru}} e : \tau$ *then* $[\![\Gamma \vdash_{\mathsf{tru}} e : \tau]\!]^{\mathsf{T}}$.

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

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 metafunction $\lfloor \cdot \rfloor$ on the corresponding definitions from §3. However, in order to make an algorithmic and rich λ^{GTL} with truer typing, we use a bidirectional type system and translation to λ^{ICTL}. This allows us to capitalize on truer typing's ability to take advantage of the tag checks from casts to refine the types of expressions.

Figure 16 presents a few salient rules of the judgments that together define the type checker for λ^{GTL} expressions and their translation to λ^{ICTL}. Unlike the translations for simple and tag, the truer typing translation produces a type τ' for the translated term e. The translation is defined so that $\tau' \leq \tau$ in the subtyping lattice, which implies that the evaluation of e in Transient must at least enforce the types of t (by the subsumption rule), but might possibly also enforce even "stronger" types. From the perspective of a language designer, both the λ^{GTL} τ-based reasoning and the λ^{ICTL} τ'-based reasoning are preserved (an example is given in § 5.3).

The "infers" judgment of the bidirectional translation ($\Gamma \vdash_{\mathsf{tru}} t \Rightarrow \tau \rightsquigarrow e' : \tau'$) 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, translates it to the λ^{ICTL} expression e, and calculates the λ^{ICTL} type τ'. However, when t contains a type annotation—such as in the return type annotation of a function—rather than inserting a cast expression that is supposed to enforce the annotation, the judgment appeals to the "checks" judgment $\Gamma \vdash_{\mathsf{tru}} t \Leftarrow^{+} \tau \rightsquigarrow e : \tau'$ that attempts to construct a translated function body e' that has type τ, and calculate its type τ'. After all, the truer type system is supposed to reflect the types that Transient 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. $\Gamma \vdash_{\mathsf{tru}} t \Leftarrow \tau \rightsquigarrow e : \tau'$ inserts an appropriate type ascription to an application or a pair projection, and delegates back to the "checks" judgment to construct a term that has the portion of τ that the ascription does not cover (see Figure 18, bottom right, for the definition of $\cdot \setminus \cdot$). $\Gamma \vdash_{\mathsf{tru}} t \Leftarrow^{\Rightarrow} \tau \rightsquigarrow e : \tau'$ 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 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^{ICTL} 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 5.8 (The Translation From λ^{GTL} to λ^{ICTL} Preserves Truer Types).
If $\Gamma \vdash_{\mathsf{tru}} t \Rightarrow \tau \rightsquigarrow e : \tau'$ *then* $\tau' \leq \tau$ *and* $\Gamma \vdash_{\mathsf{tru}} e : \tau'$.

And as described, this result can be composed with subsumption to get a theorem for λ^{GTL} types:

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

With translation in hand, the type preserving theorem for truer typing can be composed with the vigilance theorem as described in §3, to get a vigilance result for λ^{GTL} expressions.

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

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

Theorem 5.11 (Natural is Not Vigilant for λ^{GTL} Truer Typing). *There are* Γ, t, e, τ *such that* $\Gamma \vdash_{\mathsf{tru}} t \Rightarrow \tau \rightsquigarrow e$ *and* $\neg[\![\Gamma \vdash_{\mathsf{tru}} e : \tau]\!]^{\mathsf{N}}$.

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. [24] use a sophisticated constraint system to infer when Transient's tag checks may be elided due to static information that the type system computes. The static information the truer type system provides may very naturally be used to implement and prove correct a similar elision pass for Transient tag checks.[10]

[10] 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 optimize the checks a function performs on its arguments, unlike Vitousek et al. [24] who do.

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

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

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

Fig. 18. Truer Typing: Check-elision optimization for λ^{ICTL}

For example, consider the variant of the example from §1 in Figure 17. Here, the snippet defines two different type adapters for the library function `segment`, with different truer types. Since calls to `png_crop` ensure a tag check on each result, `segment_png_small` will produce PNGs, while `segment_png` will not.

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

Fig. 17. Two Type Adapters for an Image Library

At each projection in `segment\_png` and `segment\_png\_small`, Transient checks that the result is a PNG. This tag check is however only necessary in the case of `segment_png`, where it is not statically known that (due to other checks) a PNG would be produced. Precisely this difference between `segment_png` and `segment_png_small`, which allows eliding a check in one case but not the other, is reflected in their truer types!

In terms of the rules of the truer type system from Figure 15, all rules that involve an expression that performs a check of a tag k strengthen the type of the expression to $K \sqcap \tau$. Hence, the tag check improves what can be statically known about the behavior of the expression in hand — rather than only knowing that it behaves according to τ, we also know that it behaves according to K. As a result, such a tag check is useful only when the strengthened type ($K \sqcap \tau \neq \tau$) is more precise than τ — that is, when it is not already known that the value in question would behave like a K ($\tau \not\leq K$).

Figure 18 provides an overview of an elision pass for redundant tag checks. The judgment $\Gamma \vdash_{\text{opt}} e : \tau \rightsquigarrow e$ consumes a typing environment and a λ^{ICTL} expression e, type checks e at τ the same way as the truer type system for λ^{ICTL}, and uses the deduced types to translate e to an equivalent expression e' without some redundant tag checks. In essence, the translation replaces a type ascription τ with $K \setminus \tau$ where K is a tag that the translated expression checks. In general, $K \setminus \tau$ denotes the tag check that is necessary to enforce K given that τ is already known — in the typing lattice, this is $*$ if $\tau \leqslant: K$, and K otherwise. The elision pass preserves contextual equivalence:

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

The proof is by defining a binary logical relation, proving it sound for contextual equivalence, and showing that the original and optimized term are logically related, and hence equivalent.

6 RELATED WORK

Type soundness, the mainstay of statically typed languages, has seen numerous interpretations in the gradually typed world. §2 discusses two different type soundness theorems and their shortcomings: the standard type soundness and tag soundness. But the heterogeneity of how the literature uses the term type soundness goes well beyond that. Chaudhuri et al. [4] prove standard type soundness

but only for fully annotated GTL programs. Muehlboeck and Tate [16] prove a type soundness theorem for a restrictive nominal gradual type system rather than a typical structural gradual type system. Tobin-Hochstadt and Felleisen [21]'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* [15, 21, 22, 27] that describes what kinds of run-time type errors a well-typed program can raise. Vitousek et al. [25] establish an "open-world" type soundness theorem for a GTL with Transient semantics that guarantees the evaluation of a well-typed program produces either a tag typed result or certain run-time errors. These properties are all variants of syntactic type soundness. Vigilance is a semantic property that goes beyond (semantic) type soundness.

As a standard for gradually typed languages, Siek et al. [20] propose the *gradual guarantees*. Even though the gradual guarantees are useful guidelines for language designers, they are orthogonal to the question of whether the semantics of an ICTL enforces the types of a GTL. The static gradual guarantee concerns only the type system. The dynamic one can be true for a semantics that enforces no types at all. We conjecture that the simple and tag type systems satisfy the static gradual guarantee, and with the Natural and Transient semantics respectively, they satisfy the dynamic gradual guarantee. However, the truer type system violates the static gradual guarantee.

As an example of how truer typing violates the static guarantee, consider the term in Figure 19. The function `foo` uses the return type annotation to insert assertions on the first and second projections, meaning the constructed pair is checked to have natural numbers as components. The function `bar` simply returns the result of applying `foo` to its argument. If we were to change the return annotation on foo to $*$, the projections would no longer have assertions for Nat. Since Transient only performs checks for tags, there is no check or assertion on the result of `bar` that would be inserted by the translation that can guarantee the type Nat×Nat, and therefore this program cannot typecheck with a return annotation of $*$ on `foo`. Fundamentally, truer typing is flow sensitive and specific to the checks and assertions that Transient inserts, and therefore seems incompatible with the static gradual guarantee as is. That said, despite not satisfying the gradual guarantees, truer typing is a useful exercise in demonstrating a power of vigilance for designers. After identifying that a semantics is more than adequate for its type system, one can use vigilance to get a stronger type system for which the semantics is still adequate. We leave uniting truer typing with the gradual guarantees as future work.

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

Fig. 19. Truer: Counterex. to static gradual guarantee

Gradual Type Theory [17] 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 designers 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 outside this combination, language designers can still rely on type annotations to make decisions about their type system, enforcement, and tooling.

Jacobs et al. [13] propose an alternative to the gradual guarantees. Specifically, they focus on preserving equivalences, requiring that the embedding of a fully statically typed subset of the GTL into the GTL be fully abstract. We leave it to future work to investigate if the design of truer typing can be tuned so we have a fully abstract embedding of fully typed terms into the truer GTL.

Abstracting Gradual Typing (AGT) [6] 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 ICTLs 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 ICTL 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 ICTL (Transient) and arrive at a type system for a GTL (truer) for which it is vigilant.

There is a significant body of work on equipping gradual type systems with blame in a correct manner [1, 9, 11, 25, 26]. Vigilance currently says nothing about blame. Vigilance is concerned with a semantics of types, both static in the form of the typing system and dynamic in the form of boundary annotations. Blame is instead concerned with the mechanisms for providing accurate errors that developers can use in debugging. We conjecture that with additional instrumentation in our logical relation[11], we can incorporate properties about blame, but leave that as future work.

7 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 [23] 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 designers and 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 a semantic property that describes a gradual type system as two components, one static and one dynamic, that work together to validate and recover incomplete type information. When the statics relies on the dynamics, but the latter does not deliver, the meaning of types becomes misleading, making type-based reasoning principles faulty. When the dynamics offers more than what the statics can capture, there is a missed opportunity to increase the strength of the statics, or decrease that of the dynamics. Vigilance is a compass for exploring this design space and finding adequate design points. For instance, in this paper we use vigilance to show that Transient is not adequate for simple typing, and that there is a missed opportunity for Transient and tag typing. The first fails vigilance which entails that the dynamic component does not live up to the expectations of the static one. The second satisfies vigilance but the dynamic component performs more checks than needed. In response, with vigilance as a guide, we perform a design exercise and examine an alternative design 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 while making sure the combination is still vigilant, and find that by doing so we can recover an optimization through conventional type-based reasoning. We hope that vigilance can more generally be a tool for identifying opportunities to further incorporate reasoning principles into the design of gradual type systems [8, 23].

REFERENCES

[1] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for All. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 201–214. https://doi.org/10.1145/1926385.1926409

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

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

[11] Namely, we conjecture adding error location information throughout our ICTL, and adjusting the Σ relation to require errors at locations informed by typing histories instead of any generic error would allow us to study blame with vigilance.

[4] 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.

[5] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *European Symposium on Programming*.

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

[7] 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

[8] 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

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

[10] 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

[11] 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

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

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

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

[15] 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

[16] 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.

[17] 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

[18] 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

[19] 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.

[20] 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

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

[22] 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.

[23] 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

[24] 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

[25] 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

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

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