JFP, pages, . ©
doi:

How to Evaluate the Semantics of Gradual Types

BEN GREENMAN
PLT @ Northeastern University e-mail: benjaminlgreenman@gmail.com

CHRISTOS DIMOULAS
PLT @ Northwestern University e-mail: chrdimo@northwestern.edu

MATTHIAS FELLEISEN
PLT @ Northeastern University e-mail: matthias@ccs.neu.edu

Abstract

The literature on gradual typing presents fundamentally different strategies for enforcing the integrity of types. This paper presents a uniform evaluation framework that characterizes the differences among the major existing and some synthetic semantics. Type system designers can now evaluate a new semantics with this framework and thus compare it with others.

Key Words: complete monitoring, blame soundness, blame completeness

1 Fork in the Road

Gradual typing comes in many forms, for example:

- *Optional* typing adds a best-effort static analysis but ignores type annotation at runtime (Bracha and Griswold, 1993; Bierman et al., 2014).
- *Transient* inserts shape checks[1] in type-checked code to guarantee that operations cannot not "go wrong" due to untyped values (Vitousek et al., 2017; Vitousek, 2019).
- *Natural* enforces types with higher-order checks and thereby ensures the full integrity of types (Siek and Taha, 2006; Tobin-Hochstadt and Felleisen, 2006).
- *Concrete* requires that every value is tagged with a type and maintains integrity with simple checks (Wrigstad et al., 2010; Muehlboeck and Tate, 2017).

In addition, researchers have proposed and implemented hybrid techniques (Greenberg, 2015; Siek et al., 2015b; Greenman and Felleisen, 2018; Bloom et al., 2009; Richards et al., 2015). An outstanding and unusual exemplar of this kind is Pyret, a language targeting the educational realm (`pyret.org`).

Each of these type-enforcement strategies picks a tradeoff among static guarantees, expressiveness, and run-time costs (section 2). If stringent constraints on untyped code are acceptable, then *concrete* offers strong and inexpensive guarantees. If the goal is to interoperate with an untyped language that does not support wrapper/proxy values, then

[1] A shape check enforces a correpondence between a top-level value constructor and the top-level constructor of a type. It generalizes the tag checks found in many runtime systems.

Table 1: Informal sketch of contributions; full results in table 2 (page 53).

	N		C		F		T		A		E
type soundness	✓		✓		✓		✓		✓		×
complete monitoring	✓		✓		×		×		×		×
blame soundness	✓		✓		✓		×		✓		✓
blame completeness	✓		✓		×		×		✓		×
error preorder	N	$\lesssim$	C	$\lesssim$	F	$\lesssim$	T	$\eqsim$	A	$\lesssim$	E

transient may offer the strongest possible guarantees. If performance is not an issue, then *natural* is the perfect choice.

Unfortunately, the literature provides little guidance to programmers and language designers on how to compare different semantics. Standard meta-theoretical tools do not articulate what is gained and lost in each tradeoff (section 3). The gradual guarantee (Siek et al., 2015c), for example, is trivially satisfied by any optionally-typed language. Simply put, the field lacks an apples-to-apples way of comparing different type-enforcement strategies and considering their implications for programmers.

This paper introduces a framework for systematically comparing the behavioral guarantees offered by different semantics of gradual typing. Because each semantics is essentially a method of enforcing static types, the comparison begins with a common mixed-typed syntax. This surface syntax is then assigned multiple semantics, each of which follows a distinct protocol for enforcing type specifications. With this semantic framework, one can directly observe the possible behaviors for a single program.

The chosen models illustrate *natural* (N), *transient* (T), *optional* (henceforth *erasure*, E), and three theoretical strategies (*co-natural* C, *forgetful* F, and *amnesic* A) that demonstrate how to fill design gaps. The comparison excludes two classes of prior work: *concrete*, because of the constraints it places on untyped code (section 2.2), and gradual languages that must analyze untyped code to interoperate with it. Our focus is on strategies that can deal with untyped code as a "dusty deck" without needing to recompile the untyped world each time a new type boundary appears.

Table 1 sketches the results of the evaluation (section 6). The six letters in the top row correspond to different type-enforcement strategies, and thus different semantics, for the common surface language. Each row introduces one discriminating property. Type soundness guarantees the validity of types in typed code. Complete monitoring guarantees that the type system moderates all boundaries between typed and untyped code—even boundaries that arise at run-time. Blame soundness ensures that when a run-time check goes wrong, the error message points to boundaries that are relevant to the problem. Blame completeness guarantees that error messages come with *all* relevant information. For both blame soundness and completeness, *relevance* is determined by an independent (axiomatic) specification that tracks values as they cross boundaries between typed and untyped code (section 4.4.1). Lastly, the error preorder compares the relative "strictness" of types between two semantics. As to be expected, Natural (N) accepts the fewest programs without raising a run-time type mismatch, and Erasure (E) accepts the greatest number of programs.

In sum, the five properties enable a uniform analysis of existing strategies and can guide the search for new strategies. Indeed, the synthetic Amnesic semantics (A) demonstrates how a semantics can fail complete monitoring but guarantee sound and complete blame.

1.1 Performance

Understanding the formal properties of gradual typing systems is only half the challenge. There is a parallel and ongoing quest to uncover the performance implications of different strategies (Bauman et al., 2017; Feltey et al., 2018; Greenman and Migeed, 2018; Greenman and Felleisen, 2018; Greenman et al., 2019b). The theoretical model presented offers some hints about relative performance (section 5.1.1), but implementations may reveal different and additional bottlenecks.

1.2 Relation to Prior Work

This paper combines two conference papers (Greenman and Felleisen, 2018; Greenman et al., 2019a) and extends them in three ways: with a survey of type-enforcement strategies (section 2), with an informal comparison of the semantics (section 5), and with additional meta-theoretic results (section 6, summarized is section 7).

1.3 Outline

Sections 2 through 5 explain the *what*, *why*, and *how* of our design-space analysis. There is a huge body of work on mixed-typed language that desperately needs organizing principles (section 2). Prior attempts to organize fall short; by contrast, the properties that frame table 1 offer an expressive and scalable basis for comparison (section 3). These properties guide an apples-to-apples method that begins with a common surface language and studies different semantics (section 4). In particular, this paper analyzes six semantics based on six ideas for enforcing static types (section 5).

Section 6 presents the six semantics and the key results. Expert readers may wish to begin there and refer back to section 5 as needed. The supplementary material contains a complete formal account of our results.

2 Assorted Behaviors by Example

There are many gradually-typed languages. Figure 1 arranges a few of their names into a rough picture of the design space. Each language enables some kind of mix between typed and untyped code. Languages marked with a star ($\star$) come with a special dynamic type, often styled as $\star$, or ?, that allows partially-defined types (Siek et al., 2015c). Technically, the type system supports implicit down-casts from the dynamic type to any other type—unlike, say, `Object` in Java. Languages marked with a cross (†) add a tailor-made type system to an untyped language, but may require types for an entire module at a time (Tobin-Hochstadt et al., 2017). Other languages satisfy different goals. To avoid confusion with

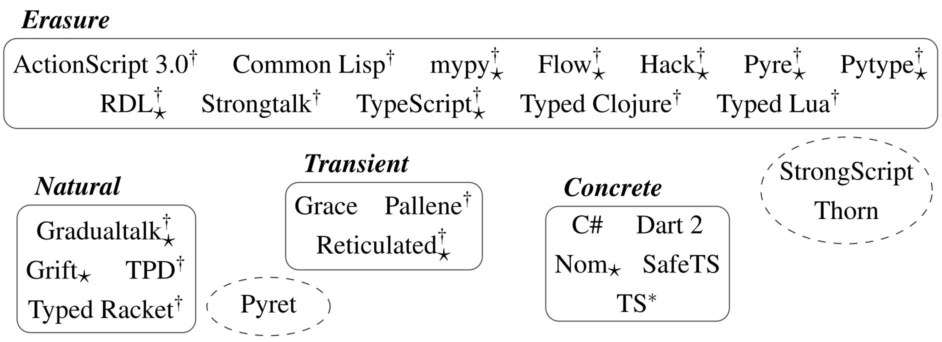

Erasure
ActionScript 3.0† Common Lisp† mypy$^\dagger_\star$ Flow$^\dagger_\star$ Hack$^\dagger_\star$ Pyre$^\dagger_\star$ Pytype$^\dagger_\star$
RDL$^\dagger_\star$ Strongtalk† TypeScript$^\dagger_\star$ Typed Clojure† Typed Lua†
Natural
Gradualtalk$^\dagger_\star$
Grift$_\star$ TPD†
Typed Racket†
Pyret
Transient
Grace Pallene†
Reticulated$^\dagger_\star$
Concrete
C# Dart 2
Nom$_\star$ SafeTS
TS*
StrongScript
Thorn

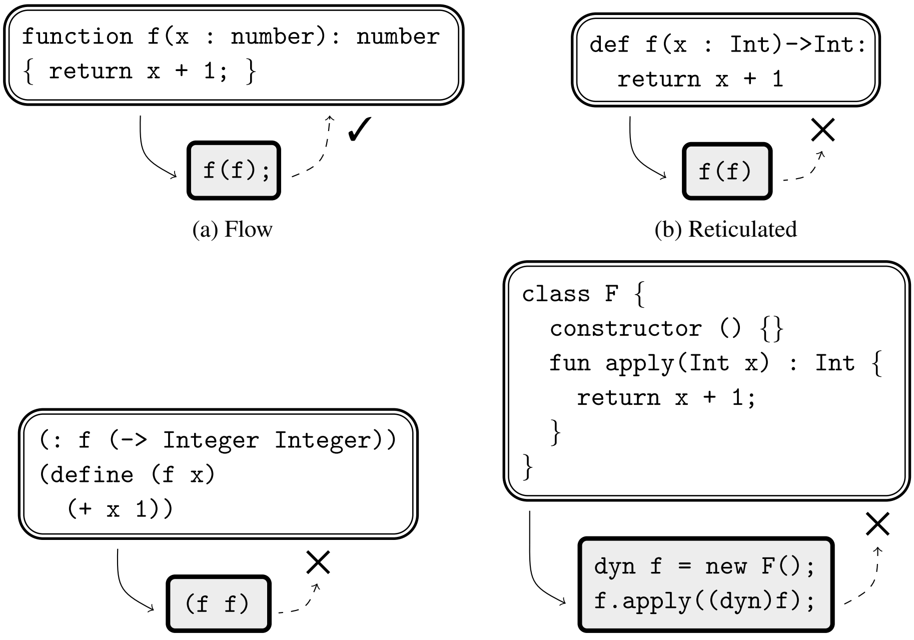
Fig. 1: Landscape of mixed-typed languages, † = migratory, ⋆ = gradual

the refined definition of *gradual typing* (Siek et al., 2015c), we use the umbrella term "mixed-typed" to describe each point in the design space.

For the most part, these mixed-typed languages fit into the broad forms introduced in section 1. Erasure is by far the most popular strategy; perhaps because of its uncomplicated semantics and ease of implementation (Steele, 1990; Ren et al., 2013; Maidl et al., 2015; Bonnaire-Sergeant et al., 2016). The Natural languages come from academic teams that are interested in types that offer strong guarantees (Tobin-Hochstadt and Felleisen, 2008; Allende et al., 2013; Williams et al., 2017; Bauman et al., 2017). Transient is gaining traction as a compromise between types and performance (Vitousek et al., 2017; Roberts et al., 2019; Gualandi and Ierusalimschy, 2020), and Concrete has generated interest among industry teams (Bierman et al., 2010; Dart, 2020) as well as academics (Swamy et al., 2014; Rastogi et al., 2015; Muehlboeck and Tate, 2017). Nevertheless, several languages explore a hybrid approach. StrongScript and Thorn offer a choice of concrete and erased types (Wrigstad et al., 2010; Richards et al., 2015). Pyret uses Natural-style checks to validate fixed-size data and Transient-style checks for recursive types (e.g. lists) and higher-order types.[2] The literature presents additional variations. Castagna and Lanvin (2017) present a Natural semantics that drops certain wrappers. Siek et al. (2015b) give a monotonic semantics that associates types with heap positions instead of creating wrappers. There are several implementation of the monotonic idea (Richards et al., 2017; Swamy et al., 2014; Rastogi et al., 2015; Almahallawi, 2020).

Our goal is a systematic comparison of type guarantees across the wide design space. Such a comparison is possible because, despite the variety, the different guarantees arise from choices about how to enforce types at the boundaries between type-checked code and arbitrary dynamically-typed code. To illustrate, the following three subsections discuss type boundary examples in the context of: Flow (Chaudhuri et al., 2017), Reticulated (Vitousek et al., 2017), Typed Racket (Tobin-Hochstadt et al., 2017), and Nom (Muehlboeck and Tate, 2017). Flow is a migratory typing system for JavaScript, Reticulated equips Python with gradual types, Typed Racket extends Racket, and Nom is a new gradual-from-the-start object-oriented language.

[2] Personal communication with Benjamin Lerner and Shriram Krishnamurthi.



```
function f(x : number): number
{ return x + 1; }
```

```
f(f);
```

(a) Flow

```
def f(x : Int)->Int:
  return x + 1
```

```
f(f)
```

(b) Reticulated

```
(: f (-> Integer Integer))
(define (f x)
  (+ x 1))
```

```
(f f)
```

(c) Typed Racket

```
class F {
  constructor () {}
  fun apply(Int x) : Int {
    return x + 1;
  }
}
```

```
dyn f = new F();
f.apply((dyn)f);
```

(d) Nom

Fig. 2: Program (2.1) translated to four languages

2.1 *Enforcing a Base Type*

One of the simplest ways that a mixed-typed interaction can go wrong is for untyped code to send incorrect input to a typed context that expects a flat value. The first example illustrates one such interaction:

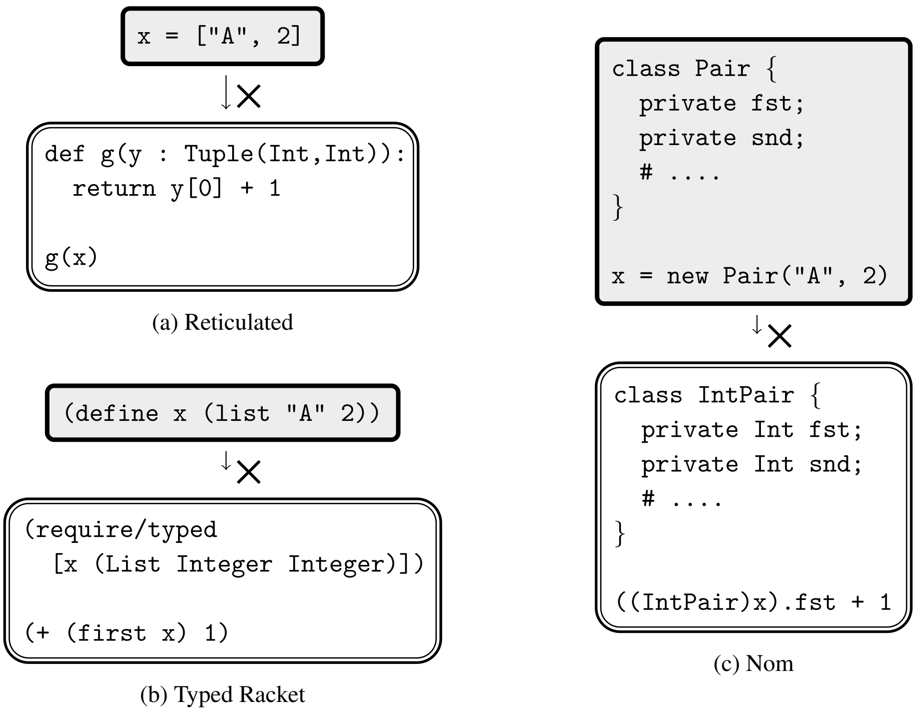
$$\begin{tikzcd} \boxed{\mathsf{f} = \lambda(\mathsf{x} : \mathsf{Int}).\, \mathsf{x} + 1} \arrow[r, bend left] & \boxed{\mathsf{f}\ \mathsf{f}} \arrow[l, bend left, dashed] \end{tikzcd} \tag{2.1}$$

The typed function on the left expects an integer. The untyped context on the right imports this function f and applies f to itself; thus the typed function receives a function rather than an integer. The question is whether the program halts or invokes the typed function f on a nonsensical input.

Figure 2 translates the program to four languages. Each white box represents type-checked code and each grey box represents untyped and, ideally, un-analyzed code that is linked in at run-time. Nom is an exception, however, because it cannot interact with truly untyped code (section 2.2). Despite the differences in syntax and types, each clearly defines a typed function that expects an integer on the top and applies the function to itself in an untyped context on the bottom.

In Flow (figure 2a), the program does not detect a type mismatch. The typed function receives a function from untyped JavaScript and surprisingly computes a string (`ECMA-262` edition 10, §12.8.3). In the other three languages, the program halts with a *boundary error* message that alerts the programmer to the mismatch between two chunks of code.

Flow does not detect the run-time type mismatch because it follows the *erasure*, or optional typing, approach to type enforcement. Erasure is hands-off; types have no effect



```
x = ["A", 2]
```

```
def g(y : Tuple(Int,Int)):
  return y[0] + 1

g(x)
```

(a) Reticulated

```
(define x (list "A" 2))
```

```
(require/typed
  [x (List Integer Integer)])

(+ (first x) 1)
```

(b) Typed Racket

```
class Pair {
  private fst;
  private snd;
  # ....
}

x = new Pair("A", 2)
```

```
class IntPair {
  private Int fst;
  private Int snd;
  # ....
}

((IntPair)x).fst + 1
```

(c) Nom

Fig. 3: Program (2.2) translations

on the behavior of a program. These static-only types help find logical mistakes and enable type-directed IDE tools, but disappear during compilation. Consequently, the author of a typed Erasure function cannot assume that it receives only well-typed input.

The other languages enforce static types with some kind of dynamic check. For base types, the check validates the shape of incoming data. The checks for other types reveal differences among these non-trivial type enforcement strategies.

2.2 Validating an Untyped Data Structure

The second example is about pairs. It asks what happens when typed code declares a pair type and receives an untyped pair:

$$\begin{tikzcd} \boxed{\mathsf{g} = \lambda(x : \mathsf{Int} \times \mathsf{Int}).\,(\mathsf{fst}\ x) + 1} \arrow[r, bend left] & \boxed{\mathsf{g}\ \langle \text{“A''}, 2 \rangle} \arrow[l, bend left, dashed] \end{tikzcd} \tag{2.2}$$

The typed function on the left expects a pair of integers and uses the first element of the input pair as a number. The untyped code on the right applies this function to a pair that contains a string and an integer.

Figure 3 translates this idea into Reticulated, Typed Racket, and Nom. The encodings in Reticulated and Typed Racket define a pair in untyped code and impose a type in typed code. The encoding in Nom is different; figure 3c presents a Nom program in which the typed code expects an instance of one data structure but the untyped code provides something else. This shape mismatch leads to a run-time error.

Nom cannot express program (2.2) directly because the language does not allow partially-typed values. There is no common pair constructor that: (1) untyped code can use without constraints and (2) typed code can instantiate at a specific type. All type structure must be specified with the data structure. On one hand, this requirement greatly simplifies run-time validation because the outermost shape of any value determines the full type of its elements. On the other hand, it imposes a significant burden on the programmer. To add refined static type checking at the use-sites of an untyped data structure, a programmer must either add a cast to each use in typed code or edit the untyped code for a new data definition. Because Nom and other concrete languages require this kind of type structure in untyped code (Wrigstad et al., 2010; Richards et al., 2015; Muehlboeck and Tate, 2017; Dart, 2020), the model in section 6 does not support them.

Both Reticulated and Typed Racket raise an error on program (2.2), but for substantially different reasons. Typed Racket rejects the untyped pair at the boundary to the typed context because the pair does not fully match the declared type. Reticulated accepts the value at the boundary because it is a pair, but raises an exception at the elimination form `y[0]` because typed code expects an integer result but receives a string. These sample behaviors are indicative of a wider difference; Typed Racket eagerly checks the contents of data structures while Reticulated lazily validates use-sites.

2.3 *Uncovering the Source of a Mismatch*

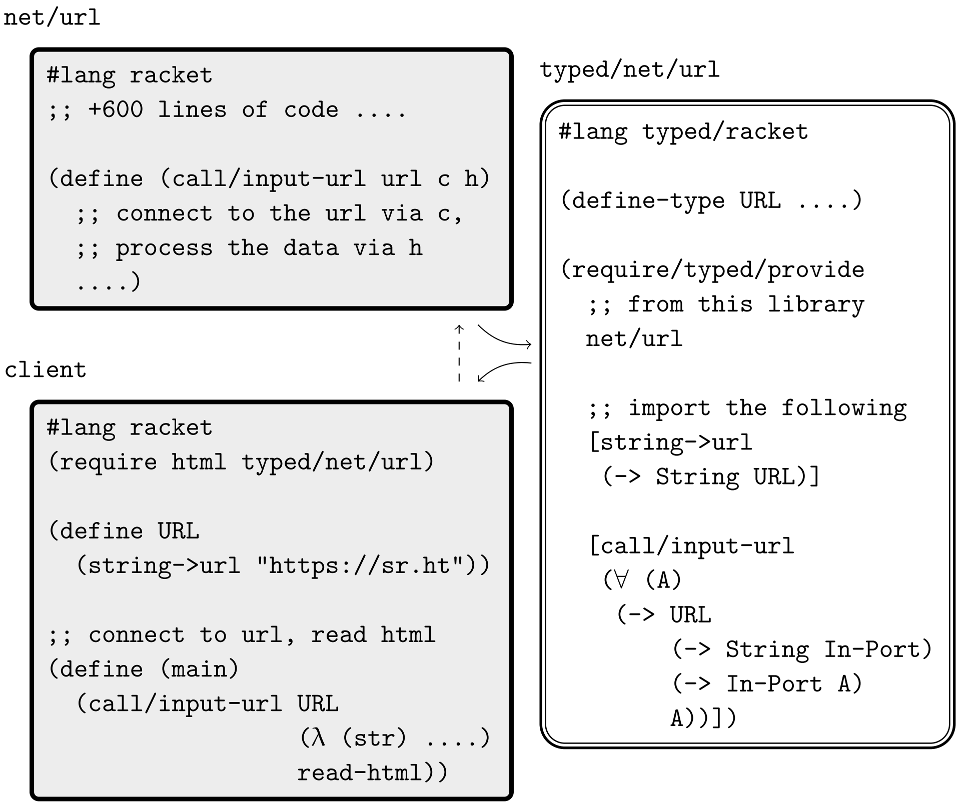
Figures 4 and 5 present excerpts from realistic programs that mix typed and untyped code. These examples follow the same general structure: an untyped client interacts with an untyped library through a thin layer of typed code. Both programs also signal run-time errors, but for different reasons and with different implications for the programmer.

Figure 4 consists of an untyped library, an *incorrect* layer of type annotations, and an untyped client of the types. The module on the top left, `net/url`, is a snippet from an untyped library that has been part of Racket for two decades.[3] The typed module on the right defines types for part of the untyped library. Lastly, the module at the bottom left imports the typed library and calls the library function `call/input-url`.

Operationally, the library function flows from `net/url` to the typed module and then to the client. When the client calls this function, it sends client data to the untyped library code via the typed module. The client application clearly relies on the type specification from `typed/net/url` because: the first argument is a URL structure, the second is a function that accepts a string, and the third is a function that maps an input port to an HTML representation. Unfortunately for the client, the type declaration in figure 4 is buggy. The library applies the first callback of `call/input-url` to a URL struct, rather than a string as the developer expects.

Fortunately for the developer, Typed Racket compiles types to contracts and thereby catches the mismatch. Here, the compilation of `typed/net/url` generates a contract for `call/input-url`. The generated contract ensures that the untyped client provides three type-matching argument values and that the library applies the callback to a string. When

[3] `github.com/racket/net`



net/url

```
#lang racket
;; +600 lines of code ....

(define (call/input-url url c h)
  ;; connect to the url via c,
  ;; process the data via h
  ....)
```

typed/net/url

```
#lang typed/racket

(define-type URL ....)

(require/typed/provide
  ;; from this library
  net/url

  ;; import the following
  [string->url
   (-> String URL)]

  [call/input-url
   (∀ (A)
    (-> URL
        (-> String In-Port)
        (-> In-Port A)
        A))])
```

client

```
#lang racket
(require html typed/net/url)

(define URL
  (string->url "https://sr.ht"))

;; connect to url, read html
(define (main)
  (call/input-url URL
                  (λ (str) ....)
                  read-html))
```

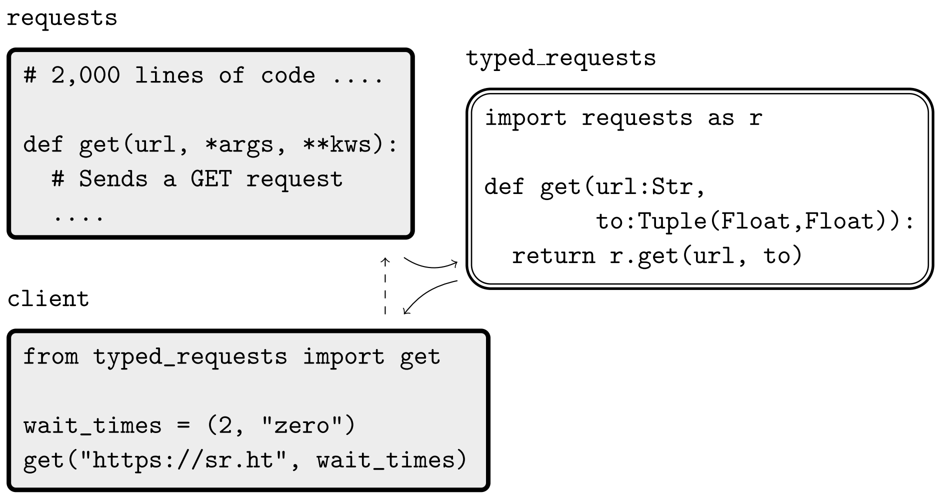
Fig. 4: Using Typed Racket to define an API

the `net/url` library eventually applies the callback function to a URL structure, the function contract for the callback halts the program. The blame message says that the interface for `net/url` broke the contract, but warns the developer on the last line with "assuming the contract is correct." A quick look confirms that the contract—that is, the type from which the contract is derived—is wrong.

Figure 5 presents an arrangement of three Transient Reticulated modules, similar to the code in figure 4. The module on the top left exports a function that retrieves data from a URL.[4] This function accepts several optional and keyword arguments. The typed adaptor module on the right formulates types for one valid use of the function; a client may supply a URL as a string and a timeout as a pair of floats. These types are correct, but the client module on the bottom left sends a tuple that contains an integer and a string.

Reticulated's runtime checks ensure that the typed function receives a string and a tuple, but do not validate the tuple's contents. These same arguments then pass to the untyped `get` function in the `requests` module. When the untyped `get` eventually uses the string `"zero"` as a float, Python raises an exception that originates from the `requests` module. A completly untyped version of this program gives the same behavior; the Reticulated types are no help for debugging.

[4] `github.com/psf/requests`



requests

```
# 2,000 lines of code ....

def get(url, *args, **kws):
  # Sends a GET request
  ....
```

typed_requests

```
import requests as r

def get(url:Str,
        to:Tuple(Float,Float)):
  return r.get(url, to)
```

client

```
from typed_requests import get

wait_times = (2, "zero")
get("https://sr.ht", wait_times)
```

Fig. 5: Using Reticulated to define an API

In this example, the programmer is lucky because the call to the typed version of get is still visible on the stack trace, providing a hint that this call might be at fault. If Python were to properly implement tail calls, or if the library accessed the pair some time after returning control to the client, this hint would disappear.

In sum, types in Transient Reticulated do not monitor all channels of communication between modules. A value may cross a type boundary without a full check, making it difficult to discover type-value mismatches or pinpoint their source. Reticulated mitigates this problem with a global map from heap addresses to source locations. The analysis in section 6 demonstrates, however, that this map can result in incorrect blame.

3 Towards a Formal Comparison

The design of a type-enforcement strategy is a multi-faceted problem. A strategy determines: whether mismatches between type specifications and value flows are discovered; whether the typed portion of the code is really statically typed, in a conventional sense; what typed APIs mean for untyped client code; and whether an error message can pinpoint which type specification does not match which value. All of these decisions imply consequences for the programmer and the language designer.

The examples in section 2 illustrate that various languages choose different points in this multi-faceted design space. But, examples can only motivate a systematic analysis; they cannot serve as the basis of such an endeavor. The selection of example programs and their translation across languages require too much insight. Worse, the examples tell us little about the broader implications of each choice; at best they can demonstrate issues.

A systematic analysis needs a suite of formal properties that capture the consequences of design choices for the working developer and language designer. Such properties must:

- apply to a wide (if not the full) spectrum of design options,
- articulate benefits of type specifications to typed and untyped code alike, and
- come with proof techniques that scale to complex language features.

The literature on gradual typing contains few adequate properties. Our analysis therefore brings new properties to the toolbox.

3.1 Comparative Properties in Prior Work

Type soundness is one formal property that meets the above criteria. A type soundness theorem can be tailored to a range of type systems, such a theorem has meaning for typed and untyped code, and the syntactic proof technique scales to a variety of language features (Wright and Felleisen, 1994). The use of type soundness in the gradual typing literature, however, does not promote a level comparison. Consider the four example languages from the previous section. Chaudhuri et al. (2017) present a model of Flow and prove a conventional type soundness theorem under the assumption that all code is statically-typed. Vitousek et al. (2017) prove a type soundness theorem for Reticulated Python; a reader will eventually notice that the theorem talks about the *shape* of values not their types. Muehlboeck and Tate (2017) prove a full type soundness theorem for Nom, which implements the concrete approach. Tobin-Hochstadt and Felleisen (2006) prove a full type soundness theorem for a prototypical Typed Racket that includes a weak blame property. To summarize, the four advertised type soundness theorems differ in several regards: one focuses on the typed half of the language; a second proves a claim about a loose relationship between values and types; a third is a truly conventional type soundness theorem; and the last one incorporates a claim about the quality of error messages.

Siek et al. (2015c) propose the *gradual guarantee* as a test to identify languages that enable smooth transitions between typed and untyped. They and others show that the gradual guarantee holds for relatively simple type languages and syntactic constructs; proving that it generalizes to complex type systems is the subject of active research (Igarashi et al., 2017; Toro et al., 2019; New et al., 2020). The guarantee itself, however, does not tell apart the behaviors in section 2. Both Reticulated and Nom come with published proofs of the gradual guarantee (Vitousek et al., 2017; Muehlboeck and Tate, 2017). Typed Racket states the guarantee as an explicit design goal. Even Flow, thanks to its lack of dynamic checks, satisfies the criteria for a smooth transition.

The KafKa framework is able to distinguish behaviors but lacks a meta-theoretical analysis (Chung et al., 2018). The sole theorem in the paper states type soundness for a statically-typed evaluation language. Different behaviors arise, however, from four translations of a mixed-typed surface language into this evaluation language. One can observe the behaviors, but the model does not characterize them.

New et al. (2019) use an analysis of equational reasoning to distinguish gradual typing systems. Starting from the β and η axioms for a typed language and additional equivalences regarding type precision ($\sqsubseteq$), they ask whether these principles are preserved when typed code interacts with untyped code. Their approach identifies differences among the various enforcement schemas. For example, the Natural semantics preserves all the equations and a lazy variant (Co-Natural) fails η for pairs. But the type-centric nature of this approach offers no direct insight to the programmer who starts with a large untyped code base and adds typed and untyped pieces to it. At best, authors of untyped code may trust that the behavior of their programs cannot be affected by certain changes in typed libraries. In contrast to the mechanical nature of a syntactic method, this approach demands novel techniques for proving that the chosen equivalences hold. Discovering such techniques is an active research area and requires ingenuity when adapting them from one linguistic setting to another.

Another well-studied property is the *blame theorem* (Tobin-Hochstadt and Felleisen, 2006; Wadler and Findler, 2009; Ahmed et al., 2011; Siek et al., 2015b; Wadler, 2015; Vitousek et al., 2017). Despite the authoritative name, this property is not the final word on blame. It states that a run-time mismatch may occur only when an untyped value enters a typed, or more-precisely typed, context; a typed value cannot trigger an error by crossing to less-typed code. The property is a useful design principle, but does not distinguish the various semantics in the literature. To its credit, the blame theorem does justify the slogan "well typed programs can't be blamed" for a Natural semantics under the assumption that all boundary types are correct. The slogan does not apply, however, to a semantics such as Transient that lets a value cross a boundary without a complete type check. Nor does it hold for incorrect types that were retroactively added to an untyped program; refer to figure 4 for one example and Dimoulas et al. (2016) for further discussion.

3.2 *Our Analysis*

The primary formal property has to be type soundness, because it tells a programmer that evaluation is well-defined in each component of a mixed-typed programs. The different levels of soundness that arise in the literature must be clearly separated, though. In addition, the canonical forms lemma that enables a proof of type soundness also enables optimizations by specifying exactly which values can arise in well-typed code.

The second property, *complete monitoring*, asks whether types guard all statically-declared and dynamically-created channels of communication between typed and untyped code. That is, every interaction between typed and untyped code is mediated by run-time checks.

When a run-time check discovers a mismatch between a type specification and a flow of values and the run-time system issues an error message, the question arises how informative the message is to a debugging programmer. *Blame soundness* and *blame completeness* ask whether a mixed-typed semantics can identify the responsible parties when a run-time type mismatch occurs. Soundness asks for a subset of the potential culprits; completeness asks for a superset.

Furthermore, the differences among type soundness theorems and the gap between type soundness and complete monitoring suggests the question how many errors an enforcement regime discovers. The answer is an *error preorder* relation, which compares semantics in terms of the run-time mismatches that they discover.

Individually, each property characterizes a particular aspect of a type-enforcement strategy. Together, the properties inform us about the nature of the multi-faceted design space that this semantics problem opens up. And in general, this work should help with the articulation of consequences of design choices for the working developer.

4 Evaluation Framework

This section introduces the basic ideas of the evaluation framework; detailed formal definitions are deferred to section 6. To formulate different type-enforcement stategies on an equal footing, the framework begins with one mixed-typed surface language (section 4.1) and models stategies as distinct semantics (section 4.2). The properties listed above support an analysis. Type soundness (section 4.3) and complete monitoring (section 4.4) characterize the type mismatches that a semantics detects. Blame soundness and blame completeness (section 4.5) measure the quality of error messages. The error preorder (section 4.6) enables direct behavioral comparisons.

4.1 Surface Language

The surface multi-language combines two independent pieces in the style of Matthews and Findler (2009). Statically-typed expressions constitute one piece; dynamically-typed expressions are the other half. Technically, these expression languages are identified by two judgments: typed expressions e_0 satisfy $\vdash e_0 : \tau_0$ for some type τ_0, and untyped expressions e_1 satisfy $\vdash e_1 : \mathcal{U}$ for the dynamic type. Boundary expressions connect the two languages syntactically and enable run-time interactions.

Note that $\mathcal{U}$ is not the flexible dynamic type that is compatible with any static type (Thatte, 1990; Siek and Taha, 2006), rather, it is the uni-type that describes all well-formed untyped expressions (Matthews and Findler, 2009). Consequently, there is no need for a type precision judgment in the surface language because all mixed-typed interactions occur through boundary expressions. How to add a dynamic type is a separate dimension that is orthogonal to the question of how to enforce types; with or without such a type, our results apply to the language's type-enforcement strategy. Whether the dynamic type is useful is a question for another time (Greenberg, 2019).

The core statically-typed (v_s) and dynamically-typed (v_d) values are mirror images, and consist of integers, natural numbers, pairs, and functions. This common set of values is the basis for typed-untyped communication. Types τ summarize values:

$$\begin{aligned} v_s &= i \mid n \mid \langle v_s, v_s \rangle \mid \lambda(x:\tau).\, e_s & \qquad \tau &= \mathsf{Int} \mid \mathsf{Nat} \mid \tau \Rightarrow \tau \mid \tau \times \tau \\ v_d &= i \mid n \mid \langle v_d, v_d \rangle \mid \lambda x.\, e_d \end{aligned}$$

These value sets are relatively small, but suffice to illustrate the behavior of gradual types for the basic ingredients of a full language. First, the values include atomic data, finite structures, and higher-order values. Second, the natural numbers n are a subset of the integers i to motivate a subtyping judgment for the typed half of the language. Subtyping helps the model distinguish between two type-sound methods of enforcing types (declaration-site vs. use-site) and demonstrates how the model can scale to include true union types, which must be part of any type system for originally-untyped code (Tobin-Hochstadt and Felleisen, 2010; Castagna and Lanvin, 2017; Tobin-Hochstadt et al., 2017).

Surface expressions include function application, primitive operations, and boundaries. The details of the first two are fairly standard (section 6.1), but note that function application comes with an explicit app operator ($\mathsf{app}\ e_0\ e_1$). Boundary expressions are the glue that enables mixed-typed programming. A program starts with named chunks of code, called components. Boundary expressions link these chunks together with a static type to describe the types of values that may cross the boundary. Suppose that a typed component named ℓ_0 imports and applies an untyped function from component ℓ_1:

$$\begin{tikzcd} \overset{\ell_1}{\boxed{\lambda x_0.\,\mathsf{sum}\ x_0\ 2}} \arrow[r, bend left, "\mathsf{Nat}\Rightarrow\mathsf{Nat}", "f"'] & \overset{\ell_0}{\boxed{f\ 9}} \end{tikzcd} \tag{4.1}$$

The surface language can model the composition of these components with a boundary expression that embeds an untyped function in a typed context. The boundary expression is annotated with a *boundary specification* $(\ell_0 \blacktriangleleft \mathsf{Nat}\Rightarrow\mathsf{Nat} \blacktriangleleft \ell_1)$ to explain that component ℓ_0 expects a function from sender ℓ_1:

$$(4.1) \;=\; \mathsf{app}\ (\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Nat}\Rightarrow\mathsf{Nat} \blacktriangleleft \ell_1)\ (\lambda x_0.\,\mathsf{sum}\ x_0\ 2))\ 9$$

In turn, this two-component expression may be imported into a larger untyped component. The sketch below shows an untyped component in the center that imports two typed components: a new typed function on the left and the expression (4.1) on the right.

$$\begin{tikzcd} \overset{\ell_3}{\boxed{\lambda(x_1 : \mathsf{Int}\times\mathsf{Int}).\,\mathsf{fst}\ x_1}} \arrow[r, bend left, "(\mathsf{Int}\times\mathsf{Int})\Rightarrow\mathsf{Int}", "g"'] & \overset{\ell_2}{\boxed{g\ x}} & (4.1) \arrow[l, bend right, "\mathsf{Nat}"', "x"] \end{tikzcd} \tag{4.2}$$

When linearized to the surface language, this term becomes:

$$\begin{aligned}(4.2) \;=\; &\mathsf{app}\ (\mathsf{stat}\ (\ell_2 \blacktriangleleft \mathsf{Int}\times\mathsf{Int}\Rightarrow\mathsf{Int} \blacktriangleleft \ell_3)\ (\lambda(x_1 : \mathsf{Int}\times\mathsf{Int}).\,\mathsf{fst}\ x_1))\\ &\qquad (\mathsf{stat}\ (\ell_2 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_0)\ (4.1))\end{aligned}$$

Technically, a boundary expression combines a boundary specification b and a sender expression. The specification includes the names of the client and sender components, in that order, along with a type to describe values that are intended to cross the boundary. Names, such as ℓ_0, come from some countable set ℓ. The boundary types guide the static type checker, but are mere suggestions unless a semantics decides to enforce them:

$$\begin{aligned} e_s &= \ldots \mid \mathsf{dyn}\ b\ e_d & \qquad b &= (\ell \blacktriangleleft \tau \blacktriangleleft \ell)\\ e_d &= \ldots \mid \mathsf{stat}\ b\ e_s & \qquad \ell &= \text{countable set of names}\end{aligned}$$

The typing judgments for typed and untyped expressions require a mutual dependence to handle boundary expressions. A well-typed expression may include any well-formed dynamically-typed code. Conversely, a well-formed untyped expression may include any typed expression that matches the specified annotation:

$$\boxed{\Gamma \vdash e : \tau} \qquad\qquad \boxed{\Gamma \vdash e : \mathcal{U}}$$

$$\frac{\Gamma_0 \vdash e_0 : \mathcal{U}}{\Gamma_0 \vdash \mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ e_0 : \tau_0} \qquad\qquad \frac{\Gamma_0 \vdash e_0 : \tau_0}{\Gamma_0 \vdash \mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ e_0 : \mathcal{U}}$$

Each surface-language component must have a name, drawn from a set ℓ of labels. These names must be *coherent* according to a judgment that validates an expression relative to a current name and a mapping from variables to names ($\mathcal{L}; \ell \Vdash e$, section 6.1). All boundary specifications must have a client name that matches the current name, and variables bound in one component cannot appear free in a different one.

The purpose of the names is to enable a notion of *ownership*, or responsibility. As an expression reduces to a value, ownership determines which components are responsible for the current expression and all subexpressions. Since component names appear in the surface syntax, they can help explain a run-time mismatch in terms of source-code boundaries. Suppose a program halts due to a mismatch between a type and a value. If one component is responsible for the value and the language can find both the client with the type expectation and source of the incompatible value, then a programmer knows exactly where to start debugging.

4.2 Semantic Framework

The surface language enables the construction of mixed-typed expressions. The next step is to assign behaviors to these programs via formal semantics that differ only in the way they enforce boundary types.

The first ingredient of a semantics is the set of result values v that expressions may reduce to. A result set typically extends the core typed and untyped values mentioned above ($v \supseteq v_s \cup v_d$). Potential reasons for the extended value set include the following:

1. to permit untyped values in typed code, and vice versa;
2. to track the identity of values on a heap;
3. to associate a value with a delayed type-check; and
4. to record the boundaries that a value has previously crossed.

Reasons 3 and 4 introduce two kinds of wrapper value.[5] A guard wrapper, written $\mathbb{G}\ b\ v$, associates a boundary specification with a value to achieve delayed type checks. A trace wrapper, written $\mathbb{T}\ \overline{b}\, v$, attaches a list of boundaries to a value as metadata. Guards are similar to boundary expressions; they separate a context component from a value component. Trace wrappers simply annotate values.

Second, a semantics must give reduction rules for boundary expressions. These rules initiate a type-enforcement strategy. For example, the Natural semantics (section 6.5) enforces full types via classic techniques (Findler and Felleisen, 2002; Matthews and Findler, 2009). It admits the following two reductions. Note a filled triangle (▶) describes a step in untyped code and an open triangle (▷) is for statically-typed code:

a – $\mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ 42 \ \blacktriangleright_{\mathsf{N}}\ 42$

b – $\mathsf{dyn}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ (\lambda x_0.\, -8) \ \triangleright_{\mathsf{N}}\ \mathbb{G}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ (\lambda x_0.\, -8)$

The first rule lets a typed number enter an untyped context. The second rule gives typed code access to an untyped function through a newly-created guard wrapper. Guard wrappers are a *higher-order* tool for enforcing higher-order types. As such, wrappers require elimination rules. The Natural semantics includes the following rule to unfold the application of a typed, guarded function into two boundaries:

c – $\mathsf{app}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ (\lambda x_0.\, -8))\ 1 \ \triangleright_{\mathsf{N}}$
$\quad\quad \mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (\mathsf{app}\ (\lambda x_0.\, -8)\ (\mathsf{stat}\ (\ell_1 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_0)\ 1))$

Other semantics have different behavior at boundaries and different supporting rules. The Transient semantics (section 6.8) takes a *first-order* approach to boundaries. Instead

[5] A language with the dynamic type will need a third wrapper for basic values that have been assigned type dynamic. We conjecture that this wrapper is the only change needed to transfer our positive results. Our negative results do not require changes for the dynamic type.

of using wrappers, it checks shapes at a boundary and guards elimination forms with shape-check expressions. For example, the following simplified reduction demonstrates a successful check:

$$\mathsf{d} - \ \mathsf{check}\{(\mathsf{Nat}\times\mathsf{Nat})\}\,\langle -1,-2\rangle \ \ \rhd_{\mathsf{T}}\ \langle -1,-2\rangle$$

The triangle is filled gray ($\rhd$) because Transient is defined via one notion of reduction that handles both typed and untyped code:

These two points, values and checking rules, are the distinctive aspects of a semantics. Other ingredients can be shared, such as the: errors, evaluation contexts, and interpretation of primitive operations. Indeed, section 6.2 defines three baseline evaluation languages—higher-order, first-order, and erasure—that abstract over the common ingredients.

4.3 Type Soundness

Type soundness asks whether evaluation is well-defined, and whether a surface-language type predicts aspects of the result. Since there are two kinds of surface expression, soundness has two parts: one for statically-typed code and one for dynamically-typed code.

For typed code, the question is whether code can trust the types of its subexpressions. If an expression with static type τ_0 reduces to a value, the question is what (if anything) the type τ_0 predicts about that value. There are a range of possible answers. At one end, the result value may match the full type τ_0 according to an evaluation-language typing judgment. The other extreme is that the result is a well-formed value of indeterminate shape. In both cases, the programmer knows that typed code cannot reach an undefined state during evaluation.

For untyped code, there is one surface type. If an expression reduces to a value, then uni-type soundness can only guarantee that the result is a well-formed value of indeterminate shape. The practical benefit of such a theorem is that untyped code cannot reach an undefined state through mixed-typed interactions.

Both parts combine into the following rough definition, where the function F and judgment $\vdash_{\mathsf{F}}$ are parameters. The function maps surface types to observations that one can make about a result; varying the choice of F offers a spectrum of soundness for typed code. The judgment $\vdash_{\mathsf{F}}$ matches a value with a description.

4.3.0.1 Definition Sketch (F-type soundness)**.**

If e_0 has static type τ_0 ($\vdash e_0 : \tau_0$), then one of the following holds:

- e_0 *reduces to a value* v_0 *and* $\vdash_{\mathsf{F}} v_0 : \mathsf{F}(\tau_0)$
- e_0 *reduces to an allowed error*
- e_0 *reduces endlessly.*

If e_0 is untyped ($\vdash e_0 : \mathcal{U}$), then one of the following holds:

- e_0 *reduces to a value* v_0 *and* $\vdash_{\mathsf{F}} v_0 : \mathcal{U}$
- e_0 *reduces to an allowed error*
- e_0 *reduces endlessly.*

4.4 Complete Monitoring

Complete monitoring tests whether a mixed-typed semantics has control over every interaction between typed and untyped code. If the property holds, then a programmer can rely on the language to run check at the proper points, for example, between the library and client demonstrated in figure 4. Concretely, if a value passes through the type $(\mathsf{Int}{\Rightarrow}\mathsf{Int})$ then complete monitoring guarantees that the language has control over every input to the function and every result that the function computes, regardless of whether these interactions occur in a typed or untyped context.

Because all such interactions originate at the boundaries between typed and untyped code, a first-draft way to formalize complete monitoring is to ask whether each boundary comes with a full run-time check when possible and an error otherwise. A language that meets this strict requirement certainly has full control. However, other good designs fail. Suppose typed code expects a pair of integers and a semantics initially admits any pair at the boundary but eventually checks that the pair contains integers. Despite the incomplete check at the boundary, this delayed-checking semantics eventually performs all necessary checks and should satisfy a complete monitoring theorem. Higher-order values raise a similar question because a single run-time check cannot prove that a function value always behaves a certain way. Nevertheless, a language that checks every call and return is in full control of the interactions between a function and its context.

Our definition of complete monitoring translates these intuitions about interactions and control into statements about *ownership labels* (Dimoulas et al., 2011). At the start of an evaluation, no interactions have occurred yet and every expression has one owner: the enclosing component. The reduction of a boundary term is the semantics of an interaction in which a value flows from one sender component to a client. At this point, the sender loses full control over the value. If the value fully matches the type expectations of the client, then the loss of control is no problem and the client gains full ownership. Otherwise, the sender and client may have to assume joint ownership of the value, depending on the nature of the reduction relation. If a semantics can create a value with multiple owners, then it admits that a component may lose full control over its interactions with other components.

Technically, an ownership label ℓ_0 names one source-code component. Expressions and values come with at least one ownership label; for example, $(42)^{\ell_0}$ is an integer with one owner and $(((42)^{\ell_0})^{\ell_1})^{\ell_2}$ is an integer with three owners, written $((42))^{\ell_0\ell_1\ell_2}$ for short. A complete monitoring theorem requires two ingredients that manage these labels. First, a reduction relation $\rightarrow^*_{\mathbf{r}}$ must propagate ownership labels to reflect interactions and checks. Second, a single-ownership judgment $\Vdash$ must test whether every value in an expression has a unique owner. To satisfy complete monitoring, reduction must preserve single-ownership.

The key single-ownership rules deal with labeled expressions and boundary terms:

$\boxed{\mathcal{L}; \ell \Vdash e}$

$$\frac{\mathcal{L}_0; \ell_0 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash (e_0)^{\ell_0}} \qquad \frac{\mathcal{L}_0; \ell_1 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash \mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ e_0}$$

Values such as $((42))^{\ell_0\ell_1}$ represent a communication that slipped through the run-time checking protocol, and therefore fail to satisfy single ownership. **Sneak Preview:** one way that a semantics can transfer a higher-order value without creating a joint-ownership is by providing controlled access through a wrapper. The client owns the wrapper, and the sender retains ownership of the enclosed value.

4.4.0.1 Definition Sketch (complete monitoring)**.**
For all $\Vdash e_0$*, any reduction* $e_0 \to^{*}_{\mathbf{r}} e_1$ *implies* $\Vdash e_1$.

The definition of complete monitoring is deceptively simple because it assumes a reduction relation that correctly propagates labels. In practice, a language comes with an unlabeled reduction relation, and it is up to a researcher to design a lifted relation that handles labeled terms. Lifting requires insight to correctly transfer labels and to ensure that labels do not change the behavior of programs. If labels do not transfer correctly, then a complete monitoring theorem becomes meaningless. And if the lifted relation depends on labels to compute a result, then a complete monitoring theorem says nothing about the original reduction relation.

4.4.1 How to lift a reduction relation

The models in section 6 present six reduction relations for a mixed-typed language. Each relation needs a lifted version to support an attempt at a complete monitoring proof. These lifted reduction relations are deferred to supplementary material, but come about semi-automatically through the following informal guidelines, or "natural laws," for labeling.

Each law describes one way that labels may be transferred or dropped during evaluation. To convey the general idea, each law also comes with a brief illustration, namely, an example reduction and a short comment. The example reductions use a hypothetical **r** relation over the surface language. Recall that stat and dyn are boundary terms; they link two components, a context and an enclosed expression, via a type. When reading an example, accept the transitions $e \;\mathbf{r}\; e$ as axioms and focus on how the labels change in response.

1. If a base value reaches a boundary with a matching base type, then the value must drop its current labels as it crosses the boundary.
$$(\mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (0)^{\ell_2\ell_1})^{\ell_0} \;\mathbf{r}\; (0)^{\ell_0}$$
The value 0 *fully matches the type* Nat.

2. Any other value that crosses a boundary must acquire the label of the new context.
$$(\mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (\langle -2, 1\rangle)^{\ell_1})^{\ell_0} \;\mathbf{r}\; \llparenthesis \langle -2, 1\rangle \rrparenthesis^{\ell_1\ell_0}$$
The pair $\langle -2, 1\rangle$ *does not match the type* Nat.

3. Every value that flows out of a value v_0 acquires the labels of v_0 and the context.
$$(\mathsf{snd}\ \llparenthesis \langle (1)^{\ell_0}, (2)^{\ell_1}\rangle \rrparenthesis^{\ell_2\ell_3})^{\ell_4} \;\mathbf{r}\; \llparenthesis 2 \rrparenthesis^{\ell_1\ell_2\ell_3\ell_4}$$
The value 2 *flows out of the pair* $\langle 1, 2\rangle$.

4. Every value that flows into a function v_0 acquires the label of the context and the reversed labels of v_0.
$$(\mathsf{app}\ \llparenthesis \lambda x_0.\, \mathsf{fst}\ x_0 \rrparenthesis^{\ell_0\ell_1}\ (\langle 8, 6\rangle)^{\ell_2})^{\ell_3} \;\mathbf{r}\; (\llparenthesis \mathsf{fst}\ \llparenthesis \langle 8, 6\rangle \rrparenthesis^{\ell_2\ell_3\ell_1\ell_0} \rrparenthesis^{\ell_0\ell_1})^{\ell_3}$$
The argument value $\langle 8, 6\rangle$ *is input to the function. The substituted body flows out of the function, and by law 3 acquires the function's labels.*

5. A primitive operation (δ) may remove labels on incoming base values.
$$(\mathsf{sum}\ (2)^{\ell_0}\ (3)^{\ell_1})^{\ell_2} \;\mathbf{r}\; (5)^{\ell_2}$$
Assuming $\delta(\mathsf{sum}, 2, 3) = 5$.

6. Consecutive equal labels may be dropped.

$$\llparenthesis 0 \rrparenthesis^{\ell_0 \ell_0 \ell_1 \ell_0} = \llparenthesis 0 \rrparenthesis^{\ell_0 \ell_1 \ell_0}$$

7. Labels on an error term may be dropped.

$$(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_1)\ (\mathsf{sum}\ 9\ (\mathsf{DivErr})^{\ell_1}))^{\ell_0}\ \mathbf{r}\ \mathsf{DivErr}$$

Note: law 4 talks about functions, but generalizes to reference cells and other values that accept input.

To show how these laws generate a lifted reduction relation, the following rules lift the examples from section 4.2. Each rule accepts input with any sequence of labels ($\overline{\ell}$), pattern-matches the important ones, and shuffles via the guidelines. The first rule (a') demonstrates a base-type boundary (law 1). The second (b') demonstrates a higher-order boundary (law 2); the new guard on the right-hand side implicitly inherits the context label. The third rule (c') sends an input (law 4) and creates new application and boundary expressions. The fourth rule (d') applies law 3 for an output.

$$\text{a'} - \ (\mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ \llparenthesis 42 \rrparenthesis^{\overline{\ell_2}})^{\ell_3}\ \blacktriangleright_{\mathsf{N}}\ (42)^{\ell_3}$$

$$\text{b'} - \ (\mathsf{dyn}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ \llparenthesis \lambda x_0 .\ \llparenthesis -8 \rrparenthesis^{\overline{\ell_2}} \rrparenthesis^{\overline{\ell_3}})^{\ell_4}\ \rhd_{\mathsf{N}}$$
$$(\mathbb{G}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ \llparenthesis \lambda x_0 .\ \llparenthesis -8 \rrparenthesis^{\overline{\ell_2}} \rrparenthesis^{\overline{\ell_3}})^{\ell_4}$$

$$\text{c'} - \ (\mathsf{app}\ \llparenthesis (\mathbb{G}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ (v_0)^{\ell_2}) \rrparenthesis^{\overline{\ell_3}}\ \llparenthesis 1 \rrparenthesis^{\overline{\ell_4}})^{\ell_5}\ \rhd_{\mathsf{N}}$$
$$(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (\mathsf{app}\ (v_0)^{\ell_2}\ (\mathsf{stat}\ (\ell_1 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_0)\ \llparenthesis 1 \rrparenthesis^{\overline{\ell_4} \ell_5 \mathit{rev}(\overline{\ell_3})}))^{\ell_2})^{\ell_5}$$

$$\text{d'} - \ (\mathsf{check}\{(\mathsf{Nat} \times \mathsf{Nat})\}\ \llparenthesis \langle \llparenthesis -1 \rrparenthesis^{\overline{\ell_0}}, \llparenthesis -2 \rrparenthesis^{\overline{\ell_1}} \rangle \rrparenthesis^{\overline{\ell_2}})^{\ell_3}\ \rhd_{\mathsf{T}}\ \llparenthesis \langle \llparenthesis -1 \rrparenthesis^{\overline{\ell_0}}, \llparenthesis -2 \rrparenthesis^{\overline{\ell_1}} \rangle \rrparenthesis^{\overline{\ell_2} \ell_3}$$

Although the design of a lifted reduction relation is a challenge for every language, the laws in this section bring across the intuition behind prior formalizations of complete monitoring (Dimoulas et al., 2011, 2012; Takikawa et al., 2012; Moore et al., 2016) and should help guide future work.

4.5 Blame Soundness, Blame Completeness

Blame soundness and blame completeness ask whether a semantics can identify the responsible parties in the event of a run-time mismatch. A type mismatch occurs when a typed context receives an unexpected value. The value may be the result of a boundary expression or an elimination form, and the underlying issue may lie with either the value, the current type expectation, or some prior communication. In any event, a programmer needs to know which components previously handled the value to begin debugging. A semantics offers information by blaming a set of boundaries (b^*); the meta-question is whether those boundaries have any connection to the value at hand.

Suppose that a reduction halts on the value v_0 and blames the set b_0^* of boundaries. Ideally, the names in these boundaries should list exactly the components that have handled this value. Ownership labels let us state the question precisely. The lifted variant of the same reduction provides an independent specification of the responsible components;

namely, the owners that get attached to v_0 as it crosses boundaries. Relative to this source-of-truth, blame soundness asks whether the names in b_0^* are a subset of the true owners. Blame completeness asks for a superset of the true owners.

A semantics can trivially satisfy blame soundness by reporting an empty set of boundaries. Conversely, the trivial way to achieve blame completeness is to blame every boundary for every possible mismatch. The real challenge is to satisfy both or implement a pragmatic tradeoff.

4.5.0.1 Definition Sketch (blame soundness)**.**
*For all reductions that end in a mismatch for value v_0 blaming boundaries b_0^*, the names in b_0^* are a **subset** of the labels on v_0.*

4.5.0.2 Definition Sketch (blame completeness)**.**
*For all reductions that end in a mismatch for value v_0 blaming boundaries b_0^*, the names in b_0^* are a **superset** of the labels on v_0.*

The propagation laws above (section 4.4.1) specify one way to manage ownership labels. But other ground-truth strategies are possible, and may provide insights about semantics that fail to be blame-sound and blame-complete with the standard labeling. As a case in point, the Transient semantics (section 6.8) uses heap addresses to allow mixed-typed interaction without wrapper expressions. The evaluation of a function, for example, draws a fresh heap address p_0 and stores the function on a value heap ($\mathcal{H}$).

$$(\lambda x_0.\, x_0);\mathcal{H}_0;\mathcal{B}_0 \;\rhd_{\mathsf{T}}\; p_0;(\{p_0 \mapsto (\lambda x_0.\, x_0)\}\cup\mathcal{H}_0);(\{p_0 \mapsto \emptyset\}\cup\mathcal{B}_0)$$
$$\text{where } p_0 \text{ fresh in } \mathcal{H}_0 \text{ and } \mathcal{B}_0$$

When this function pointer p_0 crosses a boundary, the semantics records the crossing on a blame heap ($\mathcal{B}$). The blame heap provides a set of boundaries if a type mismatch occurs, but this set is typically unsound because it conflates different pointers to the same value. Propagating labels onto the heap, however, enables a conjecture that Transient blames only boundaries that are relevant to the address of the incompatible value.

4.6 Error Preorder

Whereas the preceding properties characterize the semantics independently of each other, an *error preorder relation* allows direct comparisons. Strategies that perform many eager run-time checks have a lower position in the order.

One semantics lies below another in this preorder, written $X \lesssim Y$, if it raises errors on at least as many well-formed input expressions. Put another way, $X \lesssim Y$ if and only if the latter reduces at least as many expressions to a result value. When two semantics agree about which expressions raise run-time errors, the notation $X \eqsim Y$ shows that they lie below one another.

4.6.0.1 Definition Sketch (error preorder $\lesssim$)**.**
$X \lesssim Y$ *iff* $\{e_0 \mid \exists v_0.\, e_0 \rightarrow_X^* v_0\} \subseteq \{e_1 \mid \exists v_1.\, e_1 \rightarrow_Y^* v_1\}$.

4.6.0.2 Definition Sketch (error equivalence $\eqsim$)**.**
$X \eqsim Y$ *iff* $X \lesssim Y$ *and* $Y \lesssim X$.

5 Overview of Type-Enforcement Strategies

To validate the expressiveness of the framework, this section models six semantics. Three semantics have been implemented for full-fledged languages: Natural, Transient, and Erasure (section 2). One, Amnesic, explores a theoretical tradeoff to improve the blame guarantees of the Transient semantics. The remaining two, Co-Natural and Forgetful, originate in prior work (Greenberg, 2015; Greenman and Felleisen, 2018) and explore the gap between the Natural and Transient strategies.

This section presents an informal overview of the type-enforcement strategies; all technical definitions and properties appear in section 6. The discussion begins with the semantics that is lowest on the error preorder (Natural) and ascends to the top (Erasure):

Natural	:	Wrap higher-order values; eagerly validate first-order values.
Co-Natural	:	Wrap higher-order and first-order values.
Forgetful	:	Wrap higher-order and first-order values, but drop inner wrappers.
Transient	:	No wrappers; check the shape of all values that appear in typed code.
Amnesic	:	Check shapes like Transient; use wrappers only to remember boundaries.
Erasure	:	No wrappers; check nothing. Do not enforce static types at runtime.

5.1 Natural

Natural strictly enforces the boundaries between typed and untyped code. Every time a typed context imports an untyped value, the value receives a comprehensive check. For first-order data, this implies a deep traversal of the incoming value. For higher-order data, a full check at the time of crossing the boundary is impossible; instead, Natural wraps the incoming value to monitor its future behavior.

More formally, when an untyped value v flows into a context that expects some value of type τ, Natural employs the type-directed validation strategy in the left column below. The strategy on the right protects a typed value from an untyped context:

5.1.0.1 Natural. dyn = dynamic to static, stat = static to dynamic

– $\mathsf{dyn}\ \mathsf{Int}\ v \rhd \cdot$ check that v is an integer	– $\mathsf{stat}\ \mathsf{Int}\ v \blacktriangleright \cdot$ check nothing
– $\mathsf{dyn}\ (\tau_0 \times \tau_1)\ v \rhd \cdot$ check that v is a tuple and recursively validate its elements	– $\mathsf{stat}\ (\tau_0 \times \tau_1)\ v \blacktriangleright \cdot$ recursively protect the elements
– $\mathsf{dyn}\ (\tau_0 \Rightarrow \tau_1)\ v \rhd \cdot$ check that v is a function and wrap v to protect higher-order inputs and validate outputs	– $\mathsf{stat}\ (\tau_0 \Rightarrow \tau_1)\ v \blacktriangleright \cdot$ wrap v to validate inputs and protect higher-order outputs

5.1.1 *Theoretical Costs, Motivation for Alternative Methods*

Implementations of the Natural approach have struggled with the performance overhead of enforcing types (Findler and Felleisen, 2002; Greenman et al., 2019b) and have inspired semantics with tighter time and space bounds (Herman et al., 2010; Siek et al., 2015a; Greenberg, 2015; Bauman et al., 2017; Feltey et al., 2018; Kuhlenschmidt et al., 2019). A glance at the sketch above suggests three sources for this overhead: *checking* that a value matches a type, the layer of *indirection* that a wrapper adds, and the *allocation* cost.

For base types and higher-order types, the cost of checking is presumably low. Testing whether a value is an integer or a function is a cheap operation in languages that support dynamic typing. Pairs, however, illustrate the potential for serious overhead. When a deeply-nested pair value reaches a boundary, Natural follows the type to conduct an eager and comprehensive check. The cost of a successful check is linear in the size of the type. In a language with recursive types—perhaps for lists—the cost is linear in the size of the incoming value.

The indirection cost grows in proportion to the number of wrappers on a value. There is no limit to the number of wrappers in Natural, so this cost can grow without bound. Indeed, the combined cost of checking and indirection can lead to exponential slowdown even in simple programs (Herman et al., 2010; Greenberg, 2015; Takikawa et al., 2015; Feltey et al., 2018; Kuhlenschmidt et al., 2019).

Lastly, creating a wrapper initializes a data structure. Creating an unbounded number of wrappers incurs a proportional cost, which may add up to a significant fraction of a program's running time.

These theoretical costs motivate the next three strategies. First, the Co-Natural strategy (section 5.2) reduces the up-front cost of checks with additional wrappers. Second, the Forgetful strategy (section 5.3) reduces indirection by keeping at most two wrappers on any value and discarding the rest. Third, the Transient strategy (section 5.4) removes wrappers altogether by enforcing a weaker invariant.

5.1.2 *Origins of the Natural strategy*

The name "Natural" comes from Matthews and Findler (2009), who use it to describe a proxy method for transporting untyped functions into a typed context. Prior works on higher-order contracts (Findler and Felleisen, 2002), remote procedure calls (Ohori and Kato, 1993), and typed foreign function interfaces (Ramsey, 2008) employ a similar method. In the gradual typing literature, this method is also called "guarded" (Vitousek et al., 2014), "behavioral" (Chung et al., 2018), and "deep" (Tunnell Wilson et al., 2018).

5.2 *Co-Natural*

The Co-Natural method checks only the shape of values at a boundary. Instead of eagerly validating the contents of a data structure, Co-Natural creates a wrapper to perform validation by need. The cost of checking at a boundary is thereby reduced to the worst-case cost of a shape check. Allocation and indirection costs may increase, however, because first-order values now receive wrappers.

5.2.0.1 Co-Natural. dyn = dynamic to static, stat = static to dynamic

- $\mathsf{dyn}\ \mathsf{Int}\ v\ \rhd\ \cdot$
  check that v is an integer
- $\mathsf{dyn}\ (\tau_0 \times \tau_1)\ v\ \rhd\ \cdot$
  check that v is a tuple and wrap v to validate projections
- $\mathsf{dyn}\ (\tau_0 \Rightarrow \tau_1)\ v\ \rhd\ \cdot$
  check that v is a function and wrap v to protect higher-order inputs and validate outputs

- $\mathsf{stat}\ \mathsf{Int}\ v\ \blacktriangleright\ \cdot$
  check nothing
- $\mathsf{stat}\ (\tau_0 \times \tau_1)\ v\ \blacktriangleright\ \cdot$
  wrap v to protect future projections
- $\mathsf{stat}\ (\tau_0 \Rightarrow \tau_1)\ v\ \blacktriangleright\ \cdot$
  wrap v to validate inputs and protect higher-order outputs

5.2.1 Origins of the Co-Natural strategy

The Co-Natural strategy introduces a small amount of laziness. By contrast to Natural, which eagerly validates immutable data structures, Co-Natural waits until an elimination form. The choice is analogous to the question of initial algebra vs. final algebra semantics for such datatypes (Wand, 1979; Cartwright, 1980; Bergstra and Tucker, 1983), hence the prefix "Co" is a reminder that some checks now happen at an opposite time. Findler et al. (2007) implement exactly the Co-Natural strategy for Racket struct contracts. Other researchers have explored variations on lazy contracts (Hinze et al., 2006; Dimoulas and Felleisen, 2011; Chitil, 2012; Degen et al., 2012); for instance, by delaying even shape checks until a computation depends on the value.

5.3 Forgetful

The goal of Forgetful is to limit the number of wrappers around a value. A key non-goal is to enforce types as compositionally-valid claims about code. Typed code can rely on the static types that it declares, but nothing more. Untyped code cannot trust type annotations because those types might never be checked.

The Forgetful strategy is to keep at most two wrappers around a value. An untyped value gets one wrapper when it enters a typed context and loses this wrapper upon exit. A typed value gets a "sticky" inner wrapper the first time it exits typed code and gains a "temporary" outer wrapper whenever it re-enters a typed context.

5.3.0.1 Forgetful. dyn = dynamic to static, stat = static to dynamic

- $\mathsf{dyn}\ \mathsf{Int}\ v\ \rhd\ \cdot$
  check that v is an integer
- $\mathsf{dyn}\ (\tau_0 \times \tau_1)\ v\ \rhd\ \cdot$
  check that v is a tuple and wrap v to validate projections

- $\mathsf{stat}\ \mathsf{Int}\ v\ \blacktriangleright\ \cdot$
  check nothing
- $\mathsf{stat}\ (\tau_0 \times \tau_1)\ v\ \blacktriangleright\ \cdot$
  if v has a wrapper, discard it; otherwise wrap v to protect projections

5.3.1 Origins of the Forgetful strategy

Greenberg (2015) introduces forgetful manifest contracts and proves type soundness; the extended version of the paper contains a detailed discussion, including the observation that forgetful types cannot support abstraction and information hiding.[6] Castagna and Lanvin (2017) present a forgetful and type sound semantics for a mixed-typed language.

In contrast to Forgetful, there are other strategies that limit the number of wrappers on a value without sacficing type guarantees (Herman et al., 2010; Greenberg, 2015; Siek et al., 2015a). These methods require a protocol for merging wrappers, whereas Forgetful is a simple and type-sound way to save time and space.

5.4 Transient

The Transient method ensures that typed code does not "go wrong" (Milner, 1978) in the sense of applying a primitive operation to a value outside its domain. Every application $(e_0\ e_1)$ in typed code can, for example, trust that the value of e_0 is a function.

Transient meets this goal without the use of wrappers or deep checks. Instead, it rewrites typed code to pre-emptively check the shape of values. Every type boundary, every typed elimination form, and every typed function gets protected with a shape check; these checks validate the output of boundaries and elimination forms, and the input to typed functions.

As the name suggests, a shape check matches the top-level constructor of a value against the top-level constructor of a type. The following table describes the checks that happen at a boundary; the extra checks in typed code perform dynamic-to-static checks.

5.4.0.1 Transient. dyn = dynamic to static, stat = static to dynamic

– $\mathsf{dyn}\ \mathsf{Int}\ v\ \triangleright\ \cdot$ check that v is an integer	– $\mathsf{stat}\ \mathsf{Int}\ v\ \blacktriangleright\ \cdot$ check nothing
– $\mathsf{dyn}\ (\tau_0 \times \tau_1)\ v\ \triangleright\ \cdot$ check that v is a pair	– $\mathsf{stat}\ (\tau_0 \times \tau_1)\ v\ \blacktriangleright\ \cdot$ check nothing
– $\mathsf{dyn}\ (\tau_0 \Rightarrow \tau_1)\ v\ \triangleright\ \cdot$ check that v is a function	– $\mathsf{stat}\ (\tau_0 \Rightarrow \tau_1)\ v\ \blacktriangleright\ \cdot$ check nothing

A more expressive language of types, however, may require deeper shape checks. For instance, a union-type check must consider each alternative in the union. Greenman and Felleisen (2018) supply additional comments and performance data regarding an implementation of Transient for the Typed Racket surface language.

In general, Transient checks add up to a greater number of run-time validation points than those that arise in a wrapper-based semantics, but their overall cost may be lower and more predictable. Static analysis can reduce the number of checks (Vitousek et al., 2019).

5.4.1 Origins of the Transient strategy

Vitousek (2019) invented Transient for Reticulated Python. The name suggests the nature of its run-time checks; transient type-enforcement enforces local assumptions in typed

[6] arxiv.org/abs/1410.2813

code, but has no long-lasting ability to influence untyped behaviors (Vitousek et al., 2014). Transient has been adapted to Typed Racket (Greenman and Felleisen, 2018) and has inspired a closely-related approach for Grace (Roberts et al., 2019).

5.5 *Amnesic*

The goal of the Amnesic semantics is to provide the same behavior as Transient but improve the error messages when a type mismatch occurs. Amnesic demonstrates that wrappers offer more than a way to detect errors; they seem essential for informative errors.

The Amnesic strategy consists of three steps: it wraps values, it discards all but three wrappers, and it keeps a record of discarded boundary specifications. When a type mismatch occurs, Amnesic presents the recorded boundaries to the programmer.

Amnesic employs both guard wrappers and *trace* wrappers. In the model, a trace wrapper records a list of boundaries that a value has previously crossed. If an untyped function enters a typed component, Amnesic wraps the function in a guard. If the function travels back to untyped code, Amnesic replaces the guard with a trace wrapper that records two boundaries. Future round-trips extend the trace. Conversely, a typed function that flows to untyped code and back N+1 times gets three wrappers: an outer guard to protect its current typed client, a middle trace to record its last N trips, and an inner guard to protect its body.

5.5.0.1 Amnesic. dyn = dynamic to static, stat = static to dynamic

- dyn Int v $\triangleright$ $\cdot$
  check that v is an integer
- dyn $(\tau_0 \times \tau_1)$ v $\triangleright$ $\cdot$
  check that v is a tuple and wrap v to check projections
- dyn $(\tau_0 \Rightarrow \tau_1)$ v $\triangleright$ $\cdot$
  check that v is a function and wrap v to protect higher-order inputs and validate outputs

- stat Int v $\blacktriangleright$ $\cdot$
  check nothing
- stat $(\tau_0 \times \tau_1)$ v $\blacktriangleright$ $\cdot$
  if v has a guard wrapper, replace with a trace; otherwise guard v
- stat $(\tau_0 \Rightarrow \tau_1)$ v $\blacktriangleright$ $\cdot$
  if v has a guard wrapper, replace with a trace; otherwise guard v

5.5.1 *Origins of the Amnesic strategy*

Amnesic is a synthesis of Forgetful and Transient that demonstrates how our framework can guide the design of new checking strategies (Greenman et al., 2019a). The name suggests a connection to forgetful, and the Greek origin of the second author.

5.6 *Erasure*

The Erasure approach is based on a view of types as an optional syntactic artifact. Type annotations are a structured form of comment that help developers and tools read a codebase. At runtime, types are meaningless. Any value may flow into any context:

5.6.0.1 Erasure. dyn = dynamic to static, stat = static to dynamic

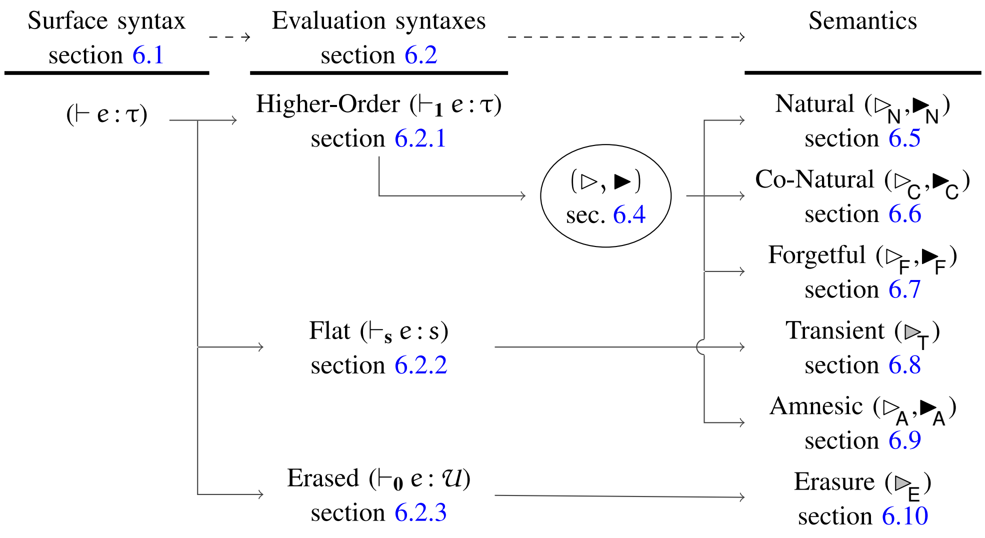

Surface syntax
section 6.1
Evaluation syntaxes
section 6.2
Semantics
$(\vdash e : \tau)$
Higher-Order $(\vdash_1 e : \tau)$
section 6.2.1
Flat $(\vdash_s e : s)$
section 6.2.2
Erased $(\vdash_0 e : \mathcal{U})$
section 6.2.3
$(\rhd, \blacktriangleright)$
sec. 6.4
Natural $(\rhd_N, \blacktriangleright_N)$
section 6.5
Co-Natural $(\rhd_C, \blacktriangleright_C)$
section 6.6
Forgetful $(\rhd_F, \blacktriangleright_F)$
section 6.7
Transient $(\rhd_T)$
section 6.8
Amnesic $(\rhd_A, \blacktriangleright_A)$
section 6.9
Erasure $(\rhd_E)$
section 6.10

Fig. 6: Map of basic definitions in section 6

- $\mathsf{dyn}\ \mathsf{Int}\ v \ \rhd\ \cdot$
  check nothing
- $\mathsf{dyn}\ (\tau_0 \times \tau_1)\ v \ \rhd\ \cdot$
  check nothing
- $\mathsf{dyn}\ (\tau_0 \Rightarrow \tau_1)\ v \ \rhd\ \cdot$
  check nothing

- $\mathsf{stat}\ \mathsf{Int}\ v \ \blacktriangleright\ \cdot$
  check nothing
- $\mathsf{stat}\ (\tau_0 \times \tau_1)\ v \ \blacktriangleright\ \cdot$
  check nothing
- $\mathsf{stat}\ (\tau_0 \Rightarrow \tau_1)\ v \ \blacktriangleright\ \cdot$
  check nothing

Despite the complete lack of type enforcement, the Erasure strategy is widely used (figure 1) and has a number of pragmatic benefits. The static type checker can point out logical errors in type-annotated code. An IDE may use the static types in auto-completion and refactoring tools. An implementation does not require any instrumentation to enforce types. Users that are familiar with the host language do not need to learn a new semantics to understand the behavior of type-annotated programs. Finally, Erasure programs run as fast as a host-language program.

5.6.1 Origins of the Erasure strategy

Erasure is also known as *optional typing* and dates back to the type hints of MACLISP (Moon, 1974) and Common Lisp (Steele, 1990). StrongTalk is another early and influential optionally-typed language (Bracha and Griswold, 1993). Models of optional typing exist for JavaScript (Bierman et al., 2014; Chaudhuri et al., 2017), Lua (Maidl et al., 2015), and Clojure (Bonnaire-Sergeant et al., 2016).

6 Technical Development

This section presents the main technical details of our analysis: the model, the six semantics, and the properties that each semantics satisfies. Because this is a long and

$\boxed{\text{Surface Syntax}}$

$$
\begin{array}{lcl@{\qquad}lcl}
e & = & x \mid i \mid n \mid \langle e, e\rangle \mid \lambda x.\, e \mid \lambda(x:\tau).\, e \mid & b & = & (\ell \blacktriangleleft \tau \blacktriangleleft \ell) \\
 & & \mathsf{app}\{{}^{\tau}\!/_{\mathcal{U}}\}\; e\; e \mid \mathit{unop}\{{}^{\tau}\!/_{\mathcal{U}}\}\; e \mid \mathit{binop}\{{}^{\tau}\!/_{\mathcal{U}}\}\; e\; e \mid & b^* & = & \mathcal{P}(b) \\
 & & \mathsf{dyn}\; b\; e \mid \mathsf{stat}\; b\; e & \ell & = & \text{countable set of names} \\
\tau & = & \mathsf{Int} \mid \mathsf{Nat} \mid \tau \Rightarrow \tau \mid \tau \times \tau & \bar{\ell} & = & \text{sequences of names} \\
{}^{\tau}\!/_{\mathcal{U}} & = & \tau \mid \mathcal{U} & \Gamma & = & \cdot \mid (x : {}^{\tau}\!/_{\mathcal{U}}), \Gamma \\
\mathit{unop} & = & \mathsf{fst} \mid \mathsf{snd} & i & = & \mathbb{Z} \\
\mathit{binop} & = & \mathsf{sum} \mid \mathsf{quotient} & n & = & \mathbb{N}
\end{array}
$$

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

$$
\frac{(x_0 : \tau_0) \in \Gamma_0}{\Gamma_0 \vdash x_0 : \tau_0}
\qquad
\frac{(x_0 : \tau_0), \Gamma_0 \vdash e_0 : \tau_1}{\Gamma_0 \vdash \lambda(x_0 : \tau_0).\, e_0 : \tau_0 \Rightarrow \tau_1}
\qquad
\frac{\Gamma_0 \vdash e_0 : \tau_1 \qquad \Delta(\mathit{unop}, \tau_1) \leqslant: \tau_0}{\Gamma_0 \vdash \mathit{unop}\{\tau_0\}\; e_0 : \tau_0}
$$

$$
\frac{\Gamma_0 \vdash e_0 : \tau_1 \Rightarrow \tau_2 \qquad \Gamma_0 \vdash e_1 : \tau_1 \qquad \tau_2 \leqslant: \tau_0}{\Gamma_0 \vdash \mathsf{app}\{\tau_0\}\; e_0\; e_1 : \tau_0}
\qquad
\frac{\Gamma_0 \vdash e_0 : \mathcal{U}}{\Gamma_0 \vdash \mathsf{dyn}\; (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; e_0 : \tau_0}
$$

$\boxed{\Gamma \vdash e : \mathcal{U}}$ selected rules

$$
\frac{(x_0 : \mathcal{U}) \in \Gamma_0}{\Gamma_0 \vdash x_0 : \mathcal{U}}
\qquad
\frac{(x_0 : \mathcal{U}), \Gamma_0 \vdash e_0 : \mathcal{U}}{\Gamma_0 \vdash \lambda x_0.\, e_0 : \mathcal{U}}
\qquad
\frac{\Gamma_0 \vdash e_0 : \tau_0}{\Gamma_0 \vdash \mathsf{stat}\; (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; e_0 : \mathcal{U}}
$$

Fig. 7: Surface syntax and typing rules

intricate section, figure 6 gives an outline. The discussion begins with one surface syntax (section 6.1) and proceeds with three target languages that can run surface programs (section 6.2). Each target comes with a target type system; type soundness relates surface types to target types. Section 6.4 presents notions of reduction that are shared among several languages. The final sections state the six base semantics and their properties.

Several properties depend on a lifted semantics that propagates ownership labels in accordance with the guidelines from section 4.4.1. This means that the map in figure 6 is only half of the formal development; each syntax and semantics has a parallel, lifted version. Section 6.1 presents the lifted surface syntax, but other sections give only the most important details regarding ownership. Full definitions appear in the supplement.

6.1 Surface Syntax, Types, and Ownership

Figure 7 presents the syntax and typing judgments for the common syntax sketched in section 4.1. Expressions e include variables, integers, pairs, functions, primitive operations, applications, and boundary expressions. The primitive operations consist of pair projections and arithmetic functions, to model interactions with a runtime system. A dyn boundary expression embeds a dynamically-typed expression into a statically-typed context, and a stat boundary expression embeds a typed expression in an untyped context.

$\boxed{\text{Ownership Syntax}}$

$$
\begin{array}{lcl}
e & = & x \mid i \mid n \mid \langle e, e\rangle \mid \lambda x.\, e \mid \lambda(x:\tau).\, e \mid \\
 & & \mathsf{app}\{{}^{\tau}\!/_{\mathcal{U}}\}\; e\; e \mid \mathit{unop}\{{}^{\tau}\!/_{\mathcal{U}}\}\; e \mid \mathit{binop}\{{}^{\tau}\!/_{\mathcal{U}}\}\; e\; e \mid \\
 & & \mathsf{dyn}\; b\; (e)^{\ell} \mid \mathsf{stat}\; b\; (e)^{\ell} \mid (e)^{\ell} \\
\ell & = & \text{countable set} \\
\mathcal{L} & = & \cdot \mid (x:\ell), \mathcal{L}
\end{array}
$$

$\boxed{e : {}^{\tau}\!/_{\mathcal{U}}\ \mathbf{wf}}$

$$
\begin{array}{l}
(e_0)^{\ell_0} : \tau_0\ \mathbf{wf} \\
\quad \text{if } \ell_0 \Vdash (e_0)^{\ell_0} \text{ and } \cdot \vdash e_0 : \tau_0 \\
(e_0)^{\ell_0} : \mathcal{U}\ \mathbf{wf} \\
\quad \text{if } \ell_0 \Vdash (e_0)^{\ell_0} \text{ and } \cdot \vdash e_0 : \mathcal{U}
\end{array}
$$

$\boxed{\mathcal{L}; \ell \Vdash e}$

$$
\frac{(x_0 : \ell_0) \in \mathcal{L}_0}{\mathcal{L}_0; \ell_0 \Vdash x_0}
\qquad
\frac{}{\mathcal{L}_0; \ell_0 \Vdash i_0}
\qquad
\frac{(x_0 : \ell_0), \mathcal{L}_0; \ell_0 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash \lambda x_0.\, e_0}
\qquad
\frac{(x_0 : \ell_0), \mathcal{L}_0; \ell_0 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash \lambda(x_0 : \tau_0).\, e_0}
$$

$$
\frac{\mathcal{L}_0; \ell_0 \Vdash e_0 \qquad \mathcal{L}_0; \ell_0 \Vdash e_1}{\mathcal{L}_0; \ell_0 \Vdash \langle e_0, e_1\rangle}
\qquad
\frac{\mathcal{L}_0; \ell_0 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash \mathit{unop}\{{}^{\tau}\!/_{\mathcal{U}}\}\; e_0}
$$

$$
\frac{\mathcal{L}_0; \ell_0 \Vdash e_0 \qquad \mathcal{L}_0; \ell_0 \Vdash e_1}{\mathcal{L}_0; \ell_0 \Vdash \mathit{binop}\{{}^{\tau}\!/_{\mathcal{U}}\}\; e_0\; e_1}
\qquad
\frac{\mathcal{L}_0; \ell_0 \Vdash e_0 \qquad \mathcal{L}_0; \ell_0 \Vdash e_1}{\mathcal{L}_0; \ell_0 \Vdash \mathsf{app}\{{}^{\tau}\!/_{\mathcal{U}}\}\; e_0\; e_1}
$$

$$
\frac{\mathcal{L}_0; \ell_1 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash \mathsf{dyn}\; (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; e_0}
\qquad
\frac{\mathcal{L}_0; \ell_1 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash \mathsf{stat}\; (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; e_0}
\qquad
\frac{\mathcal{L}_0; \ell_0 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash (e_0)^{\ell_0}}
$$

Fig. 8: Ownership syntax and single-owner consistency

A type specification ${}^{\tau}\!/_{\mathcal{U}}$ is either a static type τ or the symbol $\mathcal{U}$ for untyped code. Fine-grained mixtures of τ and $\mathcal{U}$, such as $\mathsf{Int} \times \mathcal{U}$, are not permitted; the model describes two parallel syntaxes that are connected through boundary expressions (section 4.1). A statically-typed expression e_0 is one where the judgment $\Gamma_0 \vdash e_0 : \tau_0$ holds for some type environment and type. This judgment depends on a standard notion of subtyping ($\leqslant:$) that is based on the relation $\mathsf{Nat} \leqslant: \mathsf{Int}$, is covariant for pairs and function codomains, and is contravariant for function domains. The metafunction Δ determines the output type of a primitive operation. For example the sum of two natural numbers is a natural ($\Delta(\mathsf{sum}, \mathsf{Nat}, \mathsf{Nat}) = \mathsf{Nat}$) but the sum of two integers returns an integer. A dynamically-typed expression e_1 is one for which $\Gamma_1 \vdash e_1 : \mathcal{U}$ holds for some environment.

Every function application and operator application comes with a type specification ${}^{\tau}\!/_{\mathcal{U}}$ for the expected result. These annotations serve two purposes: to determine the behavior of the Transient and Amnesic semantics, and to disambiguate statically-typed and dynamically-typed redexes. An implementation could easily infer valid annotations. The model keeps them explicit to easily formulate examples where subtyping affects behavior; for instance, the terms $\mathit{unop}\{\mathsf{Nat}\}\; e_0$ and $\mathit{unop}\{\mathsf{Int}\}\; e_0$ may give different results for the same input expression.

Figure 8 extends the surface syntax with ownership labels and introduces a single-owner ownership consistency relation. These labels record the component from which an expression originates. The extended syntax brings one addition, labeled expressions $(e)^{\ell}$, and a requirement that boundary expressions label their inner component. The single-owner consistency judgment ($\mathcal{L}; \ell \Vdash e$) ensures that every subterm of an expression has a unique owner. This judgment is parameterized by a mapping from variables to labels ($\mathcal{L}$) and a context label (ℓ). Every variable reference must occur in a context that matches the variable's map entry, every labeled expression must match the context, and every boundary expressions must have a client name that matches the context label. For example, the expression $(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (x_0)^{\ell_1})^{\ell_0}$ is consistent under a mapping that contains $(x_0 : \ell_1)$ and the ℓ_0 context label. The expression $((42)^{\ell_0})^{\ell_1}$, also written $(\!(42)\!)^{\ell_0 \ell_1}$ (figure 10), is inconsistent for any parameters.

Labels correspond to component names but come from a distinct set. Thus the expression $(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (x_0)^{\ell_1})$ contains two names, ℓ_0 and ℓ_1, and one label $^{\ell_1}$ that matches the inner component name. The distinction separates an implementation from a specification. A semantics, or implementation, manipulates component names to explain errors. Labels serve as a specification to assess whether a semantics uses component names in a sensible way. If the two could mix, then the specification would be a biased measure.

Lastly, a surface expression is well-formed ($e : {}^{\mathcal{T}}\!/\!_{\mathcal{U}}\ \mathbf{wf}$) if it satisfies a typing judgment—either static or dynamic—and single-owner consistency under some labeling and context label ℓ_0. The theorems below all require well-formed expressions.

6.2 Three Evaluation Syntaxes

Each semantics requires a unique evaluation syntax, but overlaps among these six languages motivate three common definitions. A *higher-order* evaluation syntax supports type-enforcement strategies that require wrappers. A *flat* syntax, with simple checks rather than wrappers, supports Transient. And an *erased* syntax supports the compilation of typed and untyped code to a common untyped host.

Figure 9 defines common aspects of the evaluation syntaxs. These include errors Err, shapes (or, constructors) s, evaluation contexts, and evaluation metafunctions.

A program evaluation may signal four kinds of errors.

- A dynamic tag error (TagErr) occurs when an untyped redex applies an elimination form to a mis-shaped input. For example, the first projection of an integer signals a tag error.
- An invariant error ($\mathsf{InvariantErr}$) occurs when the shape of a typed redex contradicts static typing; a "tag error" in typed code is one way to reach an invariant error. One goal of type soundness is to eliminate such contradictions.
- A division-by-zero error (DivErr) may be raised by an application of the $\mathsf{quotient}$ primitive; a full language will contain many similar primitive errors.
- A boundary error ($\mathsf{BoundaryErr}\,(b^*, v)$) reports a mismatch between two components. One component, the sender, provided the enclosed value. A second component rejected the value. The set of witness boundaries suggests potential sources for the fault; intuitively, this set should include the client–sender boundary. The error $\mathsf{BoundaryErr}\,(\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}, v_0)$, for example, says that a mismatch between value v_0 and type τ_0 prevented the value sent by the ℓ_1 component from entering the ℓ_0 component.

$\boxed{\text{Common Evaluation Syntax}}$ extends Surface Syntax

$$
\begin{array}{lcl}
\mathsf{Err} & = & \mathsf{TagErr} \mid \mathsf{InvariantErr} \mid \mathsf{DivErr} \mid \mathsf{BoundaryErr}\,(b^*, v) \\
e & = & \ldots \mid \mathsf{Err} \\
s & = & \mathsf{Int} \mid \mathsf{Nat} \mid \mathsf{Pair} \mid \mathsf{Fun} \\
E & = & [\,] \mid \mathsf{app}\{{}^{\tau}\!/\!_{\mathcal{U}}\}\;E\;e \mid \mathsf{app}\{{}^{\tau}\!/\!_{\mathcal{U}}\}\;v\;E \mid \langle E, e\rangle \mid \langle v, E\rangle \mid \mathit{unop}\{{}^{\tau}\!/\!_{\mathcal{U}}\}\;E \mid \mathit{binop}\{{}^{\tau}\!/\!_{\mathcal{U}}\}\;E\;v \mid \\
 & & \mathit{binop}\{{}^{\tau}\!/\!_{\mathcal{U}}\}\;v\;E \mid \mathsf{dyn}\;b\;E \mid \mathsf{stat}\;b\;E
\end{array}
$$

$$
\lfloor \tau_0 \rfloor = \begin{cases} \mathsf{Nat} & \text{if } \tau_0 = \mathsf{Nat} \\ \mathsf{Int} & \text{if } \tau_0 = \mathsf{Int} \\ \mathsf{Pair} & \text{if } \tau_0 \in \tau{\times}\tau \\ \mathsf{Fun} & \text{if } \tau_0 \in \tau{\Rightarrow}\tau \end{cases}
$$

$$
\mathit{shape\text{-}match}\,(s_0, v_0) = \begin{cases}
\mathsf{True} \\
\quad \text{if } s_0 = \mathsf{Nat} \text{ and } v_0 \in n \\
\quad \text{or } s_0 = \mathsf{Int} \text{ and } v_0 \in i \\
\quad \text{or } s_0 = \mathsf{Pair} \text{ and} \\
\qquad v_0 \in \langle v, v\rangle \cup \\
\qquad\quad (\mathbb{G}\;(\ell\blacktriangleleft(\tau{\times}\tau)\blacktriangleleft\ell)\;v) \\
\quad \text{or } s_0 = \mathsf{Fun} \text{ and} \\
\qquad v_0 \in (\lambda x.\,e) \cup (\lambda(x:\tau).\,e) \cup \\
\qquad\quad (\mathbb{G}\;(\ell\blacktriangleleft(\tau{\Rightarrow}\tau)\blacktriangleleft\ell)\;v) \\
\mathit{shape\text{-}match}\,(s_0, v_1) \\
\quad \text{if } v_0 = \mathbb{T}\;b_0^*\;v_1 \\
\mathsf{False} \\
\quad \text{otherwise}
\end{cases}
$$

$$
\delta(\mathit{unop}, \langle v_0, v_1\rangle) = \begin{cases} v_0 & \text{if } \mathit{unop} = \mathsf{fst}\{{}^{\tau}\!/\!_{\mathcal{U}}\} \\ v_1 & \text{if } \mathit{unop} = \mathsf{snd}\{{}^{\tau}\!/\!_{\mathcal{U}}\} \end{cases}
$$

$$
\delta(\mathit{binop}, i_0, i_1) = \begin{cases}
i_0 + i_1 \\
\quad \text{if } \mathit{binop} = \mathsf{sum}\{{}^{\tau}\!/\!_{\mathcal{U}}\} \\
\mathsf{DivErr} \\
\quad \text{if } \mathit{binop} = \mathsf{quotient}\{{}^{\tau}\!/\!_{\mathcal{U}}\} \\
\quad \text{and } i_1 = 0 \\
\lfloor i_0 / i_1 \rfloor \\
\quad \text{if } \mathit{binop} = \mathsf{quotient}\{{}^{\tau}\!/\!_{\mathcal{U}}\} \\
\quad \text{and } i_1 \neq 0
\end{cases}
$$

Fig. 9: Common evaluation syntax and metafunctions

$$
\mathit{rev}\,(b_0^*) = \{(\ell_1\blacktriangleleft\tau_0\blacktriangleleft\ell_0) \mid (\ell_0\blacktriangleleft\tau_0\blacktriangleleft\ell_1) \in b_0^*\}
\qquad
\mathit{rev}\,(\ell_0 \cdots \ell_n) = \ell_n \cdots \ell_0
$$

$$
\mathit{senders}\,(b_0^*) = \{\ell_1 \mid (\ell_0\blacktriangleleft\tau_0\blacktriangleleft\ell_1) \in b_0^*\}
\qquad
\mathit{owners}\,(v_0) = \begin{cases} \{\ell_0\} \cup \mathit{owners}\,(v_1) & \text{if } v_0 = (v_1)^{\ell_0} \\ \mathit{owners}\,(v_1) & \text{if } v_0 = \mathbb{T}\;b_0^*\;v_1 \\ \{\} & \text{otherwise} \end{cases}
$$

$$
(\!(e_0)\!)^{\ell_n \cdots \ell_1} = e_1 \quad\iff\quad e_1 = (\cdots(e_0)^{\ell_n}\cdots)^{\ell_1}
$$

Fig. 10: Metafunctions for boundaries and labels

The four shapes, s, correspond both to type constructors and to value constructors. Half of the correpondence is defined by the $\lfloor\cdot\rfloor$ metafunction, which maps a type to a shape. The *shape-match* metafunction is the other half; it checks the top-level shape of a value.

Both metafunctions use an $\cdot \in \cdot$ judgment, which holds if a value is a member of a set. The claim $v_0 \in n$, for example, holds when the value v_0 is a member of the set of natural

$\boxed{\text{Higher-Order Evaluation Syntax}}$ extends Common Evaluation Syntax

$$e \;=\; \ldots \,|\, \mathsf{trace}\ b^*\ e$$
$$v \;=\; i \,|\, n \,|\, \langle v, v\rangle \,|\, \lambda x.\, e \,|\, \lambda(x:\tau).\, e \,|\, \mathbb{G}\ b\ v \,|\, \mathbb{T}\ b^*\ v$$

$\boxed{\Gamma \vdash_{\mathbf{1}} e : \tau}$ selected rules, extends $\Gamma \vdash e : \tau$

$$\frac{\Gamma_0 \vdash_{\mathbf{1}} v_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathbf{1}} \mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 : \tau_0} \qquad \frac{}{\Gamma_0 \vdash_{\mathbf{1}} \mathsf{Err} : \tau_0}$$

$\boxed{\Gamma \vdash_{\mathbf{1}} e : \mathcal{U}}$ selected rules, extends $\Gamma \vdash e : \mathcal{U}$

$$\frac{\Gamma_0 \vdash_{\mathbf{1}} v_0 : \tau_0}{\Gamma_0 \vdash_{\mathbf{1}} \mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 : \mathcal{U}} \qquad \frac{\Gamma_0 \vdash_{\mathbf{1}} v_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathbf{1}} \mathbb{T}\ b_0^*\ v_0 : \mathcal{U}} \qquad \frac{}{\Gamma_0 \vdash_{\mathbf{1}} \mathsf{Err} : \mathcal{U}}$$

$\boxed{\mathcal{L}; \ell \Vdash e}$ selected rules, extends $\mathcal{L}; \ell \Vdash e$

$$\frac{\mathcal{L}_0; \ell_1 \Vdash v_0}{\mathcal{L}_0; \ell_0 \Vdash \mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0} \qquad \frac{\mathcal{L}_0; \ell_0 \Vdash v_0}{\mathcal{L}_0; \ell_0 \Vdash \mathbb{T}\ b_0^*\ v_0}$$

Fig. 11: Higher-Order syntax, typing rules, and ownership consistency

numbers. By convention, a variable without a subscript typically refers to a set and a term containing a set describes a comprehension. The term $(\lambda x.\, v)$, for instance, describes the set $\{(\lambda x_i.\, v_j) \mid x_i \in x \wedge v_j \in v\}$ of all functions that return some immediate value.

The *shape-match* metafunction also makes reference to two value constructors unique to the higher-order evaluation syntax: guard $(\mathbb{G}\ b\ v)$ and trace $(\mathbb{T}\ b^*\ v)$ wrappers. A guard has a shape determined by the type in its boundary. A trace is metadata, so *shape-match* looks past it. Section 4.2 informally justifies the design, and figure 11 formally introduces these wrapper values.

The final components of figure 9 are the δ metafunctions. These provide a standard and partial specification of the primitive operations.

Figure 10 defines additional metafunctions for boundaries and ownership labels. For boundaries, *rev* flips every client and sender name in a set of specifications. Both Transient and Amnesic reverse boundaries at function calls. The *senders* metafunction extracts the sender names from the right-hand side of every boundary specification in a set. For labels, *rev* reverses a sequence. The *owners* metafunction collects the labels around an unlabeled value stripped of any trace-wrapper metadata. Guard wrappers are not stripped because they represent boundaries. Lastly, the abbreviation $(\!(\cdot)\!)^{\overline{\cdot}}$ captures a list of boundaries. The term $(\!(4)\!)^{\ell_0\ell_1}$ is short for $(\!(4)^{\ell_0})^{\ell_1}$ and $(\!(5)\!)^{\overline{\ell}_0}$ matches 5 with $\overline{\ell}_0$ bound to the empty list.

6.2.1 *Higher-Order Syntax, Path-Based Ownership Consistency*

The higher-order evaluation syntax (figure 11) introduces the two wrapper values described in section 4.2. A guard wrapper $(\mathbb{G}\ (\ell \blacktriangleleft \tau \blacktriangleleft \ell)\ v)$ represents a boundary between two components.[7] A trace wrapper $(\mathbb{T}\ b^*\ v)$ attaches metadata to a value.

Type-enforcement strategies typically use guard wrappers to constrain the behavior of a value. For example, the Co-Natural semantics wraps any pair that crosses a boundary with a guard; this wrapper validates the elements of the pair upon future projections. Trace wrappers do not constrain behavior. A traced value simply comes with extra information; namely, a collection of the boundaries that the value has previously crossed.

The higher-order typing judgments, $\Gamma \vdash_{\mathbf{1}} e : {}^{\tau}\!/_{\mathcal{U}}$, extend the surface typing judgments with rules for wrappers and errors. Guard wrappers may appear in both typed and untyped code; the rules in each case mirror those for boundary expressions. Trace wrappers may only appear in untyped code; this restriction simplifies the Amnesic semantics (figure 20). A traced expression is well-formed iff the enclosed value is well-formed. An error term is well-typed in any context.

Figure 11 also extends the single-owner consistency judgment to handle wrapped values. For a guard wrapper, the outer client name must match the context and the enclosed value must be single-owner consistent with the inner sender name. For a trace wrapper, the inner value must be single-owner consistent relative to the context label.

6.2.2 *Flat Syntax*

The flat syntax (figure 12) supports wrapper-free gradual typing. A new expression form, $(\mathsf{check}\{{}^{\tau}\!/_{\mathcal{U}}\}\ e\ p)$, represents a shape check. The intended meaning is that the given type must match the value of the enclosed expression. If not, then the location p may be the source of the fault. Locations are names for the pairs and functions in a program. These names map to pre-values in a heap ($\mathcal{H}$) and, more importantly, to sets of boundaries in a blame map ($\mathcal{B}$). Pairs and functions are now second-class pre-values (w) that must be allocated before they may be used.

Three meta-functions define heap operations: $\cdot(\cdot)$, $\cdot[\cdot \mapsto \cdot]$, and $\cdot[\cdot \cup \cdot]$. The first gets an item from a finite map, the second replaces a blame heap entry, and the third extends a blame heap entry. Because maps are sets, set union suffices to add new entries.

The flat typing judgments check the top-level shape (s) of an expression and the well-formedness of any subexpressions. These judgments rely on a store typing ($\mathcal{T}$) to describe heap-allocated values. These types must be consistent with the actual values on the heap, a standard technical device that is spelled out in the supplement. Untyped functions may appear in a typed context and vice-versa—because there are no wrappers to enforce a separation. Shape-check expressions are valid in typed and untyped contexts.

6.2.3 *Erased Syntax*

Figure 13 defines an evaluation syntax for type-erased programs. Expressions include error terms; the typing judgment holds for any expression without free variables. Aside from

[7] Correction note: our prior work uses the name *monitor wrapper* and value constructor mon (Greenman and Felleisen, 2018; Greenman et al., 2019a). The name *guard wrapper* better matches earlier work (Dimoulas et al., 2012; Takikawa et al., 2012), in which `mon` constructs an expression and `G` constructs a wrapper.

$\boxed{\text{Flat Evaluation Syntax}}$ extends Common Evaluation Syntax

$$
\begin{array}{lcl}
e & = & \ldots \mid \mathsf{p} \mid \mathsf{check}\{{}^{\tau}\!/_{\mathcal{U}}\}\, e\ \mathsf{p} \\
v & = & i \mid n \mid \mathsf{p} \\
\mathsf{w} & = & \lambda x.\, e \mid \lambda(x:\tau).\, e \mid \langle v, v\rangle \\
\mathsf{p} & = & \text{countable set of heap locations} \\
\mathcal{H} & = & \mathcal{P}((\mathsf{p} \mapsto \mathsf{w})) \\
\mathcal{B} & = & \mathcal{P}((\mathsf{p} \mapsto b^{*})) \\
\mathcal{T} & = & \cdot \mid (\mathsf{p}:s), \mathcal{T}
\end{array}
$$

$$
\mathcal{H}_0(v_0) = \begin{cases} \mathsf{w}_0 & \text{if } v_0 \in \mathsf{p} \text{ and } (v_0 \mapsto \mathsf{w}_0) \in \mathcal{H}_0 \\ v_0 & \text{if } v_0 \notin \mathsf{p} \end{cases}
$$

$$
\mathcal{B}_0(v_0) = \begin{cases} b_0^{*} & \text{if } v_0 \in \mathsf{p} \text{ and } (v_0 \mapsto b_0^{*}) \in \mathcal{B}_0 \\ \emptyset & \text{otherwise} \end{cases}
$$

$$
\mathcal{B}_0[v_0 \mapsto b_0^{*}] = \begin{cases} \{v_0 \mapsto b_0^{*}\} \cup (\mathcal{B}_0 \setminus (v_0 \mapsto b_1^{*})) & \\ \qquad \text{if } v_0 \in \mathsf{p} \text{ and } (v_0 \mapsto b_1^{*}) \in \mathcal{B}_0 & \\ \mathcal{B}_0 \quad \text{otherwise} & \end{cases}
$$

$$
\mathcal{B}_0[v_0 \cup b_0^{*}] = \mathcal{B}_0[v_0 \mapsto b_0^{*} \cup \mathcal{B}_0(v_0)]
$$

$\boxed{\mathcal{T};\Gamma \vdash_{\mathbf{s}} e : s}$ selected rules

$$
\frac{(\mathsf{p}_0 : s_0) \in \mathcal{T}_0}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} \mathsf{p}_0 : s_0}
\qquad
\frac{(x_0 : \tau_0) \in \Gamma_0}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} x_0 : \lfloor \tau_0 \rfloor}
\qquad
\frac{\mathcal{T}_0; (x_0 : \mathcal{U}), \Gamma_0 \vdash_{\mathbf{s}} e_0 : \mathcal{U}}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} \lambda x_0.\, e_0 : \mathsf{Fun}}
$$

$$
\frac{\mathcal{T}_0; (x_0 : \tau_0), \Gamma_0 \vdash_{\mathbf{s}} e_0 : s_0}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} \lambda(x_0 : \tau_0).\, e_0 : \mathsf{Fun}}
\qquad
\frac{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} e_0 : \mathsf{Fun} \qquad \mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} e_1 : s_0}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} \mathsf{app}\{\tau_0\}\, e_0\ e_1 : \lfloor \tau_0 \rfloor}
$$

$$
\frac{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} e_0 : \mathsf{Pair}}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} \mathit{unop}\{\tau_0\}\, e_0 : \lfloor \tau_0 \rfloor}
\qquad
\frac{\mathcal{T}_0;\Gamma_0 \vdash e_0 : \mathcal{U}}{\mathcal{T}_0;\Gamma_0 \vdash \mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ e_0 : \lfloor \tau_0 \rfloor}
$$

$$
\frac{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} e_0 : s_0}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} \mathsf{check}\{\tau_0\}\, e_0\ \mathsf{p}_0 : \lfloor \tau_0 \rfloor}
\qquad
\frac{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} e_0 : \mathcal{U}}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} \mathsf{check}\{\tau_0\}\, e_0\ \mathsf{p}_0 : \lfloor \tau_0 \rfloor}
$$

$\boxed{\mathcal{T};\Gamma \vdash_{\mathbf{s}} e : \mathcal{U}}$ selected rules

$$
\frac{(\mathsf{p}_0 : s_0) \in \mathcal{T}_0}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} \mathsf{p}_0 : \mathcal{U}}
\qquad
\frac{(x_0 : \mathcal{U}) \in \Gamma_0}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} x_0 : \mathcal{U}}
\qquad
\frac{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} e_0 : \lfloor \tau_0 \rfloor}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} \mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ e_0 : \mathcal{U}}
$$

$$
\frac{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} e_0 : \mathcal{U}}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} \mathsf{check}\{\mathcal{U}\}\, e_0\ \mathsf{p}_0 : \mathcal{U}}
\qquad
\frac{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} e_0 : s_0}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathbf{s}} \mathsf{check}\{\mathcal{U}\}\, e_0\ \mathsf{p}_0 : \mathcal{U}}
$$

Fig. 12: Flat syntax and typing rules

the type annotations left over from the surface syntax, which could be removed with a translation step, the result is a conventional dynamically-typed language.

$\boxed{\text{Erased Evaluation Syntax}}$ extends Common Evaluation Syntax

$v \;=\; i \mid n \mid \langle v, v\rangle \mid \lambda x.\, e \mid \lambda(x\!:\!\tau).\, e$

$\boxed{\Gamma \vdash_{\mathbf{0}} e : \mathcal{U}}$ selected rules

$$\frac{(x_0 : {}^{\tau}\!/\!_{\mathcal{U}}) \in \Gamma_0}{\Gamma_0 \vdash_{\mathbf{0}} x_0 : \mathcal{U}} \qquad \frac{(x_0 : \mathcal{U}), \Gamma_0 \vdash_{\mathbf{0}} e_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathbf{0}} \lambda x_0.\, e_0 : \mathcal{U}} \qquad \frac{(x_0 : \tau_0), \Gamma_0 \vdash_{\mathbf{0}} e_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathbf{0}} \lambda(x_0 : \tau_0).\, e_0 : \mathcal{U}}$$

$$\frac{\Gamma_0 \vdash_{\mathbf{0}} e_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathbf{0}} \mathsf{dyn}\ b_0\ e_0 : \mathcal{U}} \qquad \frac{\Gamma_0 \vdash_{\mathbf{0}} e_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathbf{0}} \mathsf{stat}\ b_0\ e_0 : \mathcal{U}}$$

Fig. 13: Erased evaluation syntax and typing

6.3 Properties of Interest

Type soundness guarantees that the evaluation of a well-formed expression (1) cannot end in an invariant error and (2) preserves an evaluation-language image of the surface type. Note that an invariant error captures the classic idea of going wrong (Milner, 1978).

Definition 6.1 (F-type soundness)**.** *Let* F *map surface types to evaluation types. A semantics* X *satisfies* $\mathbf{TS}(F)$ *if for all* $e_0 : {}^{\tau}\!/\!_{\mathcal{U}}$ **wf** *one of the following holds:*

- $e_0 \rightarrow^*_X v_0$ *and* $\vdash_F v_0 : F({}^{\tau}\!/\!_{\mathcal{U}})$
- $e_0 \rightarrow^*_X \{\mathsf{TagErr}, \mathsf{DivErr}\} \cup \mathsf{BoundaryErr}\,(b^*, v)$
- $e_0 \rightarrow^*_X$ *diverges.*

Three surface-to-evaluation maps (F) suffice for the evaluation languages: an identity map **1**, a type-shape map **s** that extends the type-to-shape metafunction from figure 9, and a constant dynamic map **0**:

$$\mathbf{1}({}^{\tau}\!/\!_{\mathcal{U}}) = {}^{\tau}\!/\!_{\mathcal{U}} \qquad \mathbf{s}({}^{\tau}\!/\!_{\mathcal{U}}) = \begin{cases} \mathcal{U} & \text{if } {}^{\tau}\!/\!_{\mathcal{U}} = \mathcal{U} \\ \lfloor \tau_0 \rfloor & \text{if } {}^{\tau}\!/\!_{\mathcal{U}} = \tau_0 \end{cases} \qquad \mathbf{0}({}^{\tau}\!/\!_{\mathcal{U}}) = \mathcal{U}$$

Complete monitoring guarantees that the type on each component boundary supervises all interactions between client and server components. The definition of “all interactions” comes from the path-based ownership propagation laws (section 4.4.1); the labels on a value enumerate all partially-responsible components. Relative to this specification, a reduction that preserves single-owner consistency (figure 8) ensures that a value cannot enter a new component without a full type check.

Definition 6.2 (complete monitoring)**.** *A semantics* X *satisfies* **CM** *if for all* $(e_0)^{\ell_0} : {}^{\tau}\!/\!_{\mathcal{U}}$ **wf** *and all* e_1 *such that* $e_0 \rightarrow^*_X e_1$*, the contractum is single-owner consistent:* $\ell_0 \Vdash e_1$.

Blame soundness and *blame completeness* measure the quality of error messages relative to a specification of the components that handled a value during an evaluation. A blame-sound semantics guarantees a subset of the true senders, though it may miss some or even all. A blame-complete semantics guarantees all the true senders, though it may include irrelevant information. A sound and complete semantics reports exactly the components that sent the value across a partially-checked boundary.

The standard definitions for blame soundness and blame completeness rely on the path-based ownership propagation laws from section 4.4.1. Relative to these laws, the definitions relate the sender names in a set of boundaries (figure 10) to the true owners of the mismatched value.

Definition 6.3 (path-based blame soundness and blame completeness)**.** *For all well-formed* e_0 *such that* $e_0 \rightarrow^*_{X} \mathsf{BoundaryErr}\,(b^*_0, v_0)$*:*

- X *satisfies* **BS** *iff senders* $(b^*_0) \subseteq$ *owners* (v_0)
- X *satisfies* **BC** *iff senders* $(b^*_0) \supseteq$ *owners* (v_0).

A second useful specification extends the propagation laws to push the owners for each location (p) onto the value heap ($\mathcal{H}$). Section 6.8 develops this idea to characterize the blame guarantees of the Transient semantics.

Lastly, the error preorder relation allows direct behavioral comparisons. If X and Y represent two strategies for type enforcement, then $X \lesssim Y$ states that the Y semantics reduces at least as many expressions to a value as the X semantics.

Definition 6.4 (error preorder)**.** $X \lesssim Y$ *iff* $e_0 \rightarrow^*_{Y} \mathsf{Err}_0$ *implies* $e_0 \rightarrow^*_{X} \mathsf{Err}_1$ *for all well-formed expressions* e_0.

If two semantics lie below one another on the error preorder, then they report type mismatches on exactly the same well-formed expressions.

Definition 6.5 (error equivalence)**.** $X \approx Y$ *iff* $X \lesssim Y$ *and* $Y \lesssim X$.

6.4 Common Higher-Order Notions of Reduction

Four of the semantics build on the higher-order evaluation syntax. In redexes that do not mix typed and untyped values, these semantics share the common behavior specified in figure 14. The rules for typed code ($\rhd$) handle basic elimination forms and raise an invariant error ($\mathsf{InvariantErr}$) for invalid input. Type soundness ensures that such errors do not occur. The rules for untyped code ($\blacktriangleright$) raise a tag error for a malformed redex. Later definitions, for example figure 15, combine relations via set union to build one large relation to accomodate all redexes. The full reduction relation is the reflexive-transitive closure of such a set.

6.5 Natural and its Properties

Figure 15 presents the values and key reduction rules for the Natural semantics. Conventional reductions handle primitives and unwrapped functions ($\blacktriangleright$ and $\rhd$, figure 14).

$\boxed{e \rhd e}$

$\mathit{unop}\{\tau_0\}\, v_0 \quad \rhd\ \mathsf{InvariantErr}$
  if $v_0 \notin (\mathbb{G}\ (\ell\blacktriangleleft(\tau{\times}\tau)\blacktriangleleft\ell)\ v)$
  and $\delta(\mathit{unop}, v_0)$ is undefined

$\mathit{unop}\{\tau_0\}\, v_0 \quad \rhd\ \delta(\mathit{unop}, v_0)$
  if $\delta(\mathit{unop}, v_0)$ is defined

$\mathit{binop}\{\tau_0\}\, v_0\, v_1 \quad \rhd\ \mathsf{InvariantErr}$
  if $\delta(\mathit{binop}, v_0, v_1)$ is undefined

$\mathit{binop}\{\tau_0\}\, v_0\, v_1 \quad \rhd\ \delta(\mathit{binop}, v_0, v_1)$
  if $\delta(\mathit{binop}, v_0, v_1)$ is defined

$\mathsf{app}\{\tau_0\}\, v_0\, v_1 \quad \rhd\ \mathsf{InvariantErr}$
  if $v_0 \notin (\lambda(x:\tau).\, e) \cup$
  $(\mathbb{G}\ (\ell\blacktriangleleft(\tau{\Rightarrow}\tau)\blacktriangleleft\ell)\ v)$

$\mathsf{app}\{\tau_0\}\, v_0\, v_1 \quad \rhd\ e_0[x_0{\leftarrow}v_1]$
  if $v_0 = (\lambda(x_0:\tau_1).\, e_0)$

$\boxed{e \blacktriangleright e}$

$\mathit{unop}\{\mathcal{U}\}\, v_0 \quad \blacktriangleright\ \mathsf{TagErr}$
  if $v_0 \notin (\mathbb{G}\ (\ell\blacktriangleleft(\tau{\times}\tau)\blacktriangleleft\ell)\ v)$
  and $\delta(\mathit{unop}, v_0)$ is undefined

$\mathit{unop}\{\mathcal{U}\}\, v_0 \quad \blacktriangleright\ \delta(\mathit{unop}, v_0)$
  if $\delta(\mathit{unop}, v_0)$ is defined

$\mathit{binop}\{\mathcal{U}\}\, v_0\, v_1 \quad \blacktriangleright\ \mathsf{TagErr}$
  if $\delta(\mathit{binop}, v_0, v_1)$ is undefined

$\mathit{binop}\{\mathcal{U}\}\, v_0\, v_1 \quad \blacktriangleright\ \delta(\mathit{binop}, v_0, v_1)$
  if $\delta(\mathit{binop}, v_0, v_1)$ is defined

$\mathsf{app}\{\mathcal{U}\}\, v_0\, v_1 \quad \blacktriangleright\ \mathsf{TagErr}$
  if $v_0 \notin (\lambda x.\, e) \cup$
  $(\mathbb{G}\ (\ell\blacktriangleleft(\tau{\Rightarrow}\tau)\blacktriangleleft\ell)\ v)$

$\mathsf{app}\{\mathcal{U}\}\, v_0\, v_1 \quad \blacktriangleright\ e_0[x_0{\leftarrow}v_1]$
  if $v_0 = (\lambda x_0.\, e_0)$

Fig. 14: Common notions of reduction for Natural, Co-Natural, Forgetful, and Amnesic

A successful Natural reduction yields either an unwrapped value or a guard-wrapped function. Guards arise when a function value reaches a function-type boundary. Thus, the possible wrapped values are drawn from the following two sets:

$$\begin{array}{rcl} v_s & = & \mathbb{G}\ (\ell\blacktriangleleft(\tau{\Rightarrow}\tau)\blacktriangleleft\ell)\ (\lambda x.\, e) \\ & | & \mathbb{G}\ (\ell\blacktriangleleft(\tau{\Rightarrow}\tau)\blacktriangleleft\ell)\ v_d \end{array} \qquad \begin{array}{rcl} v_d & = & \mathbb{G}\ (\ell\blacktriangleleft(\tau{\Rightarrow}\tau)\blacktriangleleft\ell)\ (\lambda(x:\tau).\, e) \\ & | & \mathbb{G}\ (\ell\blacktriangleleft(\tau{\Rightarrow}\tau)\blacktriangleleft\ell)\ v_s \end{array}$$

The presented reduction rules are those relevant to the Natural strategy for enforcing static types. When a dynamically-typed value reaches a typed context (dyn), Natural checks the shape of the value against the type. If the type and value match, Natural wraps functions and recursively checks the elements of a pair. Otherwise, Natural raises an error at the current boundary. When a wrapped function receives an argument, Natural creates two new boundaries: one to protect the input to the inner, untyped function and one to validate the result.

Reduction in dynamically-typed code ($\blacktriangleright_{\mathsf{N}}$) follows a dual strategy. The rules for stat boundaries wrap functions and recursively protect the contents of pairs. The application of a wrapped function creates boundaries to validate the input to a typed function and to protect the result.

Unsurprisingly, this checking protocol ensures the validity of types in typed code and the well-formedness of expressions in untyped code. The Natural approach additionally keeps boundary types honest throughout the execution.

Theorem 6.6. *Natural satisfies* $\mathbf{TS}(\mathbf{1})$.

Proof By progress and preservation lemmas for the higher-order typing judgment ($\vdash_{\mathbf{1}}$). For example, if an untyped pair reaches a boundary then a typed step ($\rhd_{\mathsf{N}}$) makes progress to either a new pair or an error. In the former case, the new pair contains two boundary expressions:

$$\mathsf{dyn}\ (\ell_0\blacktriangleleft\tau_0{\times}\tau_1\blacktriangleleft\ell_1)\ \langle v_0, v_1\rangle\ \rhd_{\mathsf{N}}\ \langle \mathsf{dyn}\ (\ell_0\blacktriangleleft\tau_0\blacktriangleleft\ell_1)\ v_0, \mathsf{dyn}\ (\ell_0\blacktriangleleft\tau_1\blacktriangleleft\ell_1)\ v_1\rangle$$

$\boxed{\text{Natural Syntax}}$ extends Higher-Order Evaluation Syntax

$v \;=\; i \mid n \mid \langle v, v\rangle \mid \lambda x.\, e \mid \lambda(x:\tau).\, e \mid \mathbb{G}\ (\ell \blacktriangleleft \tau \Rightarrow \tau \blacktriangleleft \ell)\ v$

$\boxed{e \rhd_{\mathsf{N}} e}$

$\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\ v_0 \quad \rhd_{\mathsf{N}} \quad \mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\ v_0$
  if *shape-match* $(\lfloor \tau_0 \Rightarrow \tau_1 \rfloor, v_0)$

$\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\ \langle v_0, v_1\rangle \quad \rhd_{\mathsf{N}} \quad \langle \mathsf{dyn}\ b_0\ v_0, \mathsf{dyn}\ b_1\ v_1\rangle$
  if *shape-match* $(\lfloor \tau_0 \times \tau_1 \rfloor, \langle v_0, v_1\rangle)$
  where $b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$ and $b_1 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)$

$\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ i_0 \quad \rhd_{\mathsf{N}} \quad i_0$
  if *shape-match* $(\lfloor \tau_0 \rfloor, i_0)$

$\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 \quad \rhd_{\mathsf{N}} \quad \mathsf{BoundaryErr}\ (\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}, v_0)$
  if $\neg$*shape-match* $(\lfloor \tau_0 \rfloor, v_0)$

$\mathsf{app}\{\tau_0\}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_1 \Rightarrow \tau_2 \blacktriangleleft \ell_1)\ v_0)\ v_1 \quad \rhd_{\mathsf{N}} \quad \mathsf{dyn}\ b_0\ (\mathsf{app}\{\mathcal{U}\}\, v_0\ (\mathsf{stat}\ b_1\ v_1))$
  where $b_0 = (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1)$ and $b_1 = (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0)$

$\boxed{e \blacktriangleright_{\mathsf{N}} e}$

$\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\ v_0 \quad \blacktriangleright_{\mathsf{N}} \quad \mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\ v_0$
  if *shape-match* $(\lfloor \tau_0 \rfloor, v_0)$

$\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\ \langle v_0, v_1\rangle \quad \blacktriangleright_{\mathsf{N}} \quad \langle \mathsf{stat}\ b_0\ v_0, \mathsf{stat}\ b_1\ v_1\rangle$
  if *shape-match* $(\lfloor \tau_0 \times \tau_1 \rfloor, \langle v_0, v_1\rangle)$
  where $b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$ and $b_1 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)$

$\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ i_0 \quad \blacktriangleright_{\mathsf{N}} \quad i_0$
  if *shape-match* $(\lfloor \tau_0 \rfloor, i_0)$

$\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 \quad \blacktriangleright_{\mathsf{N}} \quad \mathsf{InvariantErr}$
  if $\neg$*shape-match* $(\lfloor \tau_0 \rfloor, v_0)$

$\mathsf{app}\{\mathcal{U}\}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\ v_0)\ v_1 \quad \blacktriangleright_{\mathsf{N}} \quad \mathsf{stat}\ b_0\ (\mathsf{app}\{\tau_1\}\, v_0\ (\mathsf{dyn}\ b_1\ v_1))$
  where $b_0 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)$ and $b_1 = (\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_0)$

$\boxed{e \rightarrow^*_{\mathsf{N}} e} \;=\; \rightarrow^*_{\mathsf{U}\{\rhd_{\mathsf{N}}, \blacktriangleright_{\mathsf{N}}, \blacktriangleright, \rhd\}}$

Fig. 15: Natural notions of reduction

The typing rules for pairs and for dyn boundaries validate the type of the result. ■

Theorem 6.7. *Natural satisfies* **CM**.

Proof By showing that a lifted variant of the $\rightarrow^*_{\mathsf{N}}$ relation preserves single-owner consistency ($\Vdash$). Full lifted rules for Natural appear in the supplementary material, but one can derive the rules by applying the guidelines from section 4.4.1. For example, consider the $\blacktriangleright_{\mathsf{N}}$ rule that wraps a function. The lifted version accepts a term with arbitrary ownership labels and propagates these labels to the result:

$$(\mathsf{stat}\ (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\ (\!(v_0)\!)^{\overline{\ell_2}})^{\ell_3} \quad \blacktriangleright_{\overline{\mathsf{N}}} \quad (\mathbb{G}\ (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\ (\!(v_0)\!)^{\overline{\ell_2}})^{\ell_3}$$
  if *shape-match* $(\lfloor \tau_0 \Rightarrow \tau_1 \rfloor, v_0)$

If the redex satisfies single-owner consistency, then the context label matches the client name ($\ell_3 = \ell_0$) and the labels inside the boundary match the sender name ($\bar{\ell}_2 = \ell_1 \cdots \ell_1$). Under these premises, the result also satisfies single-owner consistency. ■

Complete monitoring implies that the Natural semantics detects every mismatch between two components—either immediately, or as soon as a function computes an incorrect result. Consequently, every mismatch is due to a single boundary. Blame soundness and completeness ask whether Natural identifies the culprit.

Lemma 6.8. *If* e_0 *is well-formed and* $\mathsf{e}_0 \rightarrow^*_{\mathsf{N}} \mathsf{BoundaryErr}\,(\mathsf{b}^*_0, v_0)$, *then senders* $(\mathsf{b}^*_0) =$ *owners* (v_0) *and furthermore* b^*_0 *contains exactly one boundary specification.*

Proof The sole Natural rule that detects a mismatch blames a single boundary:

$$\begin{aligned}(\mathsf{e}_0)^{\ell_0} &\rightarrow^*_{\mathsf{N}} \mathsf{E}[\mathsf{dyn}\ (\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\ v_0] \\ &\rightarrow^*_{\mathsf{N}} \mathsf{BoundaryErr}\,(\{(\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\}, v_0)\end{aligned}$$

Thus $\mathsf{b}^*_0 = \{(\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\}$ and *senders* $(\mathsf{b}^*_0) = \{\ell_2\}$. This boundary is the correct one to blame only if it matches the true owner of the value; that is, *owners* $(v_0) = \{\ell_2\}$. Complete monitoring guarantees a match via $\ell_0 \Vdash \mathsf{E}[\mathsf{dyn}\ (\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\ v_0]$. ■

Corollary 6.9. *Natural satisfies* **BS** *and* **BC**.

6.6 Co-Natural and its Properties

Figure 16 presents the Co-Natural strategy. Co-Natural is a "lazy" variant of the Natural approach. Instead of eagerly validating pairs at a boundary, Co-Natural creates a wrapper to delay element-checks until they are needed.

Relative to Natural, there are two changes in the notions of reduction. First, the rules for a pair value at a pair-type boundary create guards. Second, new projection rules handle guarded pairs; these rules make a new boundary to validate the projected element.

Co-Natural still satisfies both a strong type soundness theorem and complete monitoring. Blame soundness and blame completeness follow from complete monitoring. Nevertheless, Co-Natural and Natural can behave differently.

Theorem 6.10. *Co-Natural satisfies* **TS**$(\mathbf{1})$.

Proof By progress and preservation lemmas for the higher-order typing judgment ($\vdash_{\mathbf{1}}$). For example, consider the rule that applies a wrapped function in a statically-typed context:

$$\begin{aligned}&\mathsf{app}\{\tau_0\}\,(\mathbb{G}\ (\ell_0 \blacktriangleleft (\tau_1 \Rightarrow \tau_2) \blacktriangleleft \ell_1)\ v_0)\ v_1 \ \rhd_{\mathsf{C}} \\ &\quad \mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1)\ (\mathsf{app}\{\mathcal{U}\}\,v_0\ (\mathsf{stat}\ (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_2)\ v_1))\end{aligned}$$

If the redex is well-typed, then v_1 has type τ_1 and the inner stat boundary is well-typed. Similar reasoning for v_0 shows that the untyped application in the result is well-typed. Thus the dyn boundary has type τ_2 which, by the types on the redex, is a subtype of τ_0. ■

Theorem 6.11. *Co-Natural satisfies* **CM**.

$\boxed{\text{Co-Natural Syntax}}$ extends Higher-Order Evaluation Syntax

$$v \;=\; i \mid n \mid \langle v, v\rangle \mid \lambda x.\, e \mid \lambda(x:\tau).\, e \mid \mathbb{G}\ (\ell \blacktriangleleft \tau \Rightarrow \tau \blacktriangleleft \ell)\ v \mid \mathbb{G}\ (\ell \blacktriangleleft \tau \times \tau \blacktriangleleft \ell)\ v$$

$\boxed{e \rhd_{\mathsf{C}} e}$

$$\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 \quad \rhd_{\mathsf{C}} \quad \mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0$$
$$\text{if } \textit{shape-match}\ (\lfloor \tau_0 \rfloor, v_0) \text{ and } v_0 \in \langle v, v\rangle \cup (\lambda x.\, e) \cup (\mathbb{G}\ b\ v)$$

$$\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ i_0 \quad \rhd_{\mathsf{C}} \quad i_0$$
$$\text{if } \textit{shape-match}\ (\lfloor \tau_0 \rfloor, i_0)$$

$$\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 \quad \rhd_{\mathsf{C}} \quad \mathsf{BoundaryErr}\ (\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}, v_0)$$
$$\text{if } \neg\textit{shape-match}\ (\lfloor \tau_0 \rfloor, v_0)$$

$$\mathsf{fst}\{\tau_0\}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_1 \times \tau_2 \blacktriangleleft \ell_1)\ v_0) \quad \rhd_{\mathsf{C}} \quad \mathsf{dyn}\ b_0\ (\mathsf{fst}\{\mathcal{U}\}\, v_0)$$
$$\text{where } b_0 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)$$

$$\mathsf{snd}\{\tau_0\}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_1 \times \tau_2 \blacktriangleleft \ell_1)\ v_0) \quad \rhd_{\mathsf{C}} \quad \mathsf{dyn}\ b_0\ (\mathsf{snd}\{\mathcal{U}\}\, v_0)$$
$$\text{where } b_0 = (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1)$$

$$\mathsf{app}\{\tau_0\}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_1 \Rightarrow \tau_2 \blacktriangleleft \ell_1)\ v_0)\ v_1 \quad \rhd_{\mathsf{C}} \quad \mathsf{dyn}\ b_0\ (\mathsf{app}\{\mathcal{U}\}\, v_0\ (\mathsf{stat}\ b_1\ v_1))$$
$$\text{where } b_0 = (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1) \text{ and } b_1 = (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0)$$

$\boxed{e \blacktriangleright_{\mathsf{C}} e}$

$$\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 \quad \blacktriangleright_{\mathsf{C}} \quad \mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0$$
$$\text{if } \textit{shape-match}\ (\lfloor \tau_0 \rfloor, v_0) \text{ and } v_0 \in \langle v, v\rangle \cup (\lambda(x:\tau).\, e) \cup (\mathbb{G}\ b\ v)$$

$$\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ i_0 \quad \blacktriangleright_{\mathsf{C}} \quad i_0$$
$$\text{if } \textit{shape-match}\ (\lfloor \tau_0 \rfloor, i_0)$$

$$\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 \quad \blacktriangleright_{\mathsf{C}} \quad \mathsf{InvariantErr}$$
$$\text{if } \neg\textit{shape-match}\ (\lfloor \tau_0 \rfloor, v_0)$$

$$\mathsf{fst}\{\mathcal{U}\}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\ v_0) \quad \blacktriangleright_{\mathsf{C}} \quad \mathsf{stat}\ b_0\ (\mathsf{fst}\{\tau_0\}\, v_0)$$
$$\text{where } b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$$

$$\mathsf{snd}\{\mathcal{U}\}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\ v_0) \quad \blacktriangleright_{\mathsf{C}} \quad \mathsf{stat}\ b_0\ (\mathsf{snd}\{\tau_1\}\, v_0)$$
$$\text{where } b_0 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)$$

$$\mathsf{app}\{\mathcal{U}\}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\ v_0)\ v_1 \quad \blacktriangleright_{\mathsf{C}} \quad \mathsf{stat}\ b_0\ (\mathsf{app}\{\tau_1\}\, v_0\ (\mathsf{dyn}\ b_1\ v_1))$$
$$\text{where } b_0 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1) \text{ and } b_1 = (\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_0)$$

$$\boxed{e \rightarrow^*_{\mathsf{C}} e} \;=\; \rightarrow^*_{\cup\{\rhd_{\mathsf{C}}, \blacktriangleright_{\mathsf{C}}, \blacktriangleright, \rhd\}}$$

Fig. 16: Co-Natural notions of reduction

Proof By preservation of single-owner consistency for the lifted $\rightarrow^*_{\mathsf{C}}$ relation. Consider the lifted rule that applies a wrapped function:

$$(\mathsf{app}\{\tau_0\}\ \llparenthesis(\mathbb{G}\ (\ell_0 \blacktriangleleft (\tau_1 \Rightarrow \tau_2) \blacktriangleleft \ell_1)\ (v_0)^{\ell_2}\rrparenthesis^{\overline{\ell}_3}\ v_1)^{\ell_4} \quad \rhd_{\overline{\mathsf{C}}}$$
$$(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1)\ (\mathsf{app}\{\mathcal{U}\}\, v_0\ (\mathsf{stat}\ (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0)\ (v_1)^{\ell_4\, \mathit{rev}\,(\overline{\ell}_3)}))^{\ell_2})^{\overline{\ell}_3 \ell_4}$$

If the redex satisfies single-owner consistency, then $\ell_0 = \overline{\ell}_3 = \ell_4$ and $\ell_1 = \ell_2$. Hence both sequences of labels in the result contain nothing but the context label ℓ_4. ■

Theorem 6.12. *Co-Natural satisfies* **BS** *and* **BC***.*

Proof By the same line of reasoning that supports Natural; refer to lemma 6.8. ■

Theorem 6.13. $\mathsf{N} \lesssim \mathsf{C}$.

Proof By a stuttering simulation between Natural and Co-Natural. Natural takes additional steps when a pair reaches a boundary because it immediately checks the contents; Co-Natural creates a guard wrapper. Co-Natural takes additional steps when eliminating a wrapped pair. The supplement defines the simulation relation.

The pair wrappers in Co-Natural imply $\mathsf{C} \not\lesssim \mathsf{N}$. Consider a statically-typed expression that imports an untyped pair with an ill-typed first element.

$$\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Nat}{\times}\mathsf{Nat} \blacktriangleleft \ell_1)\ \langle -2, 2\rangle$$

Natural detects the mismatch at the boundary, but Co-Natural will only raise an error if the first element is accessed. ■

6.7 Forgetful and its Properties

The Forgetful semantics (figure 17) creates wrappers to enforce pair and function types, but strictly limits the number of wrappers on any one value. An untyped value acquires at most one wrapper. A typed value acquires at most two wrappers: one to protect itself from inputs, and a second to reflect the expectations of its current client:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
v_s & = & \mathbb{G}\,b\,\langle v, v\rangle & v_d & = & \mathbb{G}\,b\,\langle v, v\rangle \\
 & \mid & \mathbb{G}\,b\,\lambda x.\,e & & \mid & \mathbb{G}\,b\,\lambda(x:\tau).\,e \\
 & \mid & \mathbb{G}\,b\,(\mathbb{G}\,b\,\langle v, v\rangle) & & & \\
 & \mid & \mathbb{G}\,b\,(\mathbb{G}\,b\,\lambda(x:\tau).\,e) & & &
\end{array}$$

Forgetful enforces this two-wrapper limit by removing the outer wrapper of any guarded value that flows to untyped code. An untyped-to-typed boundary always makes a new wrapper, but these wrappers do not accumulate because a value cannot enter typed code twice in a row; it must first exit typed code and lose one wrapper.

Removing outer wrappers does not affect the type soundness of untyped code; all well-formed values match $\mathcal{U}$, with or without wrappers. Type soundness for typed code is guaranteed by the temporary outer wrappers. Complete monitoring is lost, however, because the removal of a wrapper creates a joint-ownership situation. Similarly, Forgetful lies above Co-Natural and Natural in the error preorder.

When a type mismatch occurs, Forgetful blames one boundary. Though sound, this one boundary is generally not enough information to find the source of the problem. So, Forgetful fails to satisfy blame completeness.

Theorem 6.14. *Forgetful satisfies* $\mathbf{TS}(\mathbf{1})$.

Proof By progress and preservation lemmas for the higher-order typing judgment ($\vdash_{\mathbf{1}}$). The most interesting proof case shows that dropping a guard wrapper does not break type soundness. Suppose that a pair v_0 with static type $\mathsf{Int}{\times}\mathsf{Int}$ crosses two boundaries and re-enters typed code at a different type.

$\boxed{\text{Forgetful Syntax}}$ extends Higher-Order Evaluation Syntax

$$v \;=\; i \mid n \mid \langle v, v\rangle \mid \lambda x.\, e \mid \lambda(x : \tau).\, e \mid \mathbb{G}\;(\ell \blacktriangleleft \tau \Rightarrow \tau \blacktriangleleft \ell)\; v \mid \mathbb{G}\;(\ell \blacktriangleleft \tau \times \tau \blacktriangleleft \ell)\; v$$

$\boxed{e \rhd_{\mathsf{F}} e}$

$$\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \quad \rhd_{\mathsf{F}} \quad \mathbb{G}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0$$

if *shape-match* $(\lfloor \tau_0 \rfloor, v_0)$ and $v_0 \in \langle v, v\rangle \cup (\lambda x.\, e) \cup (\mathbb{G}\; b\; v)$

$$\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \quad \rhd_{\mathsf{F}} \quad i_0$$

if *shape-match* $(\lfloor \tau_0 \rfloor, v_0)$

$$\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \quad \rhd_{\mathsf{F}} \quad \mathsf{BoundaryErr}\,(\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}, v_0)$$

if $\neg$*shape-match* $(\lfloor \tau_0 \rfloor, v_0)$

$$\mathsf{fst}\{\tau_0\}\;(\mathbb{G}\;(\ell_0 \blacktriangleleft \tau_1 \times \tau_2 \blacktriangleleft \ell_1)\; v_0) \quad \rhd_{\mathsf{F}} \quad \mathsf{dyn}\; b_0\;(\mathsf{fst}\{\mathcal{U}\}\, v_0)$$

where $b_0 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)$

$$\mathsf{snd}\{\tau_0\}\;(\mathbb{G}\;(\ell_0 \blacktriangleleft \tau_1 \times \tau_2 \blacktriangleleft \ell_1)\; v_0) \quad \rhd_{\mathsf{F}} \quad \mathsf{dyn}\; b_0\;(\mathsf{snd}\{\mathcal{U}\}\, v_0)$$

where $b_0 = (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1)$

$$\mathsf{app}\{\tau_0\}\;(\mathbb{G}\;(\ell_0 \blacktriangleleft \tau_1 \times \tau_2 \blacktriangleleft \ell_1)\; v_0)\; v_1 \quad \rhd_{\mathsf{F}} \quad \mathsf{dyn}\; b_0\;(\mathsf{app}\{\mathcal{U}\}\, v_0\;(\mathsf{stat}\; b_1\; v_1))$$

where $b_0 = (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1)$ and $b_1 = (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0)$

$\boxed{e \blacktriangleright_{\mathsf{F}} e}$

$$\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \quad \blacktriangleright_{\mathsf{F}} \quad \mathbb{G}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0$$

if *shape-match* $(\lfloor \tau_0 \rfloor, v_0)$ and $v_0 \in \langle v, v\rangle \cup (\lambda(x : \tau).\, e)$

$$\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\;(\mathbb{G}\; b_1\; v_0) \quad \blacktriangleright_{\mathsf{F}} \quad v_0$$

if *shape-match* $(\lfloor \tau_0 \rfloor, v_0)$
and $v_0 \in \langle v, v\rangle \cup (\lambda x.\, e) \cup (\mathbb{G}\; b\; \langle v, v\rangle) \cup (\mathbb{G}\; b\; (\lambda(x : \tau).\, e))$

$$\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; i_0 \quad \blacktriangleright_{\mathsf{F}} \quad i_0$$

if *shape-match* $(\lfloor \tau_0 \rfloor, i_0)$

$$\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \quad \blacktriangleright_{\mathsf{F}} \quad \mathsf{InvariantErr}$$

if $\neg$*shape-match* $(\lfloor \tau_0 \rfloor, v_0)$

$$\mathsf{fst}\{\mathcal{U}\}\;(\mathbb{G}\;(\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\; v_0) \quad \blacktriangleright_{\mathsf{F}} \quad \mathsf{stat}\; b_0\;(\mathsf{fst}\{\tau_0\}\, v_0)$$

where $b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$

$$\mathsf{snd}\{\mathcal{U}\}\;(\mathbb{G}\;(\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\; v_0) \quad \blacktriangleright_{\mathsf{F}} \quad \mathsf{stat}\; b_0\;(\mathsf{snd}\{\tau_1\}\, v_0)$$

where $b_0 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)$

$$\mathsf{app}\{\mathcal{U}\}\;(\mathbb{G}\;(\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\; v_0)\; v_1 \quad \blacktriangleright_{\mathsf{F}} \quad \mathsf{stat}\; b_0\;(\mathsf{app}\{\tau_1\}\, v_0\;(\mathsf{dyn}\; b_1\; v_1))$$

where $b_0 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)$ and $b_1 = (\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_0)$

$\boxed{e \rightarrow^*_{\mathsf{F}} e} \;=\; \rightarrow^*_{\cup\{\rhd_{\mathsf{F}}, \blacktriangleright_{\mathsf{F}}, \blacktriangleright, \rhd\}}$

Fig. 17: Forgetful notions of reduction

$$\mathsf{dyn}\;(\ell_0 \blacktriangleleft (\mathsf{Nat} \times \mathsf{Nat}) \blacktriangleleft \ell_1)\;(\mathsf{stat}\;(\ell_1 \blacktriangleleft \mathsf{Int} \times \mathsf{Int} \blacktriangleleft \ell_2)\; v_0) \;\rightarrow^*_{\mathsf{F}}$$
$$\mathbb{G}\;(\ell_0 \blacktriangleleft (\mathsf{Nat} \times \mathsf{Nat}) \blacktriangleleft \ell_1)\;(\mathbb{G}\;(\ell_1 \blacktriangleleft \mathsf{Int} \times \mathsf{Int} \blacktriangleleft \ell_2)\; v_0)$$

No matter what value v_0 is, the result is well-typed because the context trusts the outer wrapper. If this double-wrapped value—call it v_2—crosses another boundary, Forgetful drops the outer wrapper. Nevertheless, the result is a sound dynamically-typed value:

$$\mathsf{stat}\ (\ell_3 \blacktriangleleft (\mathsf{Nat}\times\mathsf{Nat}) \blacktriangleleft \ell_0)\ v_2 \ \rightarrow^*_{\mathsf{F}}$$
$$\quad \mathbb{G}\ (\ell_1 \blacktriangleleft \mathsf{Int}\times\mathsf{Int} \blacktriangleleft \ell_2)\ v_0$$

When this single-wrapped wrapped pair reenters a typed context, it again gains a wrapper to document the context's expectation:

$$\mathsf{dyn}\ (\ell_4 \blacktriangleleft (\tau_1 \times \tau_2) \blacktriangleleft \ell_3)\ (\mathbb{G}\ (\ell_1 \blacktriangleleft \mathsf{Int}\times\mathsf{Int} \blacktriangleleft \ell_2)\ v_0) \ \rightarrow^*_{\mathsf{F}}$$
$$\quad \mathbb{G}\ (\ell_4 \blacktriangleleft (\tau_1 \times \tau_2) \blacktriangleleft \ell_3)\ (\mathbb{G}\ (\ell_1 \blacktriangleleft \mathsf{Int}\times\mathsf{Int} \blacktriangleleft \ell_2)\ v_0)$$

The new wrapper preserves soundness. ■

Theorem 6.15. *Forgetful does not satisfy* **CM**.

Proof Consider the lifted variant of the stat rule that removes an outer guard wrapper:

$$(\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ (\!(\mathbb{G}\ b_1\ v_0)\!)^{\bar{\ell}_2})^{\ell_3} \ \blacktriangleright_{\overline{\mathsf{F}}}\ (\!(v_0)\!)^{\bar{\ell}_2 \ell_3}$$
$$\quad \text{if } \textit{shape-match}\ (\lfloor \tau_0 \rfloor, (\mathbb{G}\ b_1\ v_0))$$

Suppose $\ell_0 \neq \ell_1$. If the redex satisfies single-owner consistency, then $\bar{\ell}_2$ contains ℓ_1 and $\ell_3 = \ell_0$. Thus the rule creates a contractum with two distinct labels. ■

Theorem 6.16. *Forgetful satisfies* **BS**.

Proof By a preservation lemma for a weakened version of the $\Vdash$ judgment, which is defined in the supplement. The judgment asks whether the owners on a value contain at least the name of the current component. Forgetful easily satisfies this invariant because the ownership guidelines (section 4.4.1) never drop an un-checked label. Thus, when a boundary error occurs:

$$\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 \ \rhd_{\mathsf{F}}\ \mathsf{BoundaryErr}\ (\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}, v_0)$$
$$\quad \text{if } \neg\textit{shape-match}\ (\lfloor \tau_0 \rfloor, v_0)$$

the sender name ℓ_1 matches one of the ownership labels on v_0. ■

Theorem 6.17. *Forgetful does not satisfy* **BC**.

Proof The proof of theorem 6.15 shows how a pair value can acquire two labels. A function can gain owners in a similar fashion, and reach an incompatible boundary:

$$\mathsf{dyn}\ (\ell_2 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_1)\ (\!(\lambda x_0.\, x_0)\!)^{\ell_0 \ell_1} \ \rhd_{\mathsf{F}}\ \mathsf{BoundaryErr}\ (\{(\ell_2 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_1)\}, (\!(\lambda x_0.\, x_0)\!)^{\ell_0 \ell_1})$$

In this example, the error does not point to component ℓ_0. ■

Theorem 6.18. $\mathsf{C} \lesssim \mathsf{F}$.

Proof By a stuttering simulation. Co-Natural can take extra steps at an elimination form to unwrap an arbitrary number of wrappers; Forgetful has at most two to unwrap. The Forgetful semantics shown above never steps ahead of Co-Natural, but the supplement presents a variant with Amnesic-style trace wrappers that does step ahead.

In the other direction, $\mathsf{F} \not\lesssim \mathsf{C}$ because Forgetful drops checks. Let:

$$e_0 = \mathsf{stat}\ b_0\ (\mathsf{dyn}\ (\ell_0 \blacktriangleleft (\mathsf{Nat}\Rightarrow\mathsf{Nat}) \blacktriangleleft \ell_1)\ (\lambda x_0.\, x_0))$$
$$e_1 = \mathsf{app}\{\mathcal{U}\}\ e_0\ \langle 2, 8 \rangle$$

Then $\mathsf{e}_1 \rightarrow^*_{\mathsf{F}} \langle 2, 8\rangle$ and Co-Natural raises a boundary error. ■

6.8 Transient and its Properties

The Transient semantics in figure 18 builds on the flat evaluation syntax (figure 12); it stores pairs and functions on a heap as indicated by the syntax of figure 12, and aims to enforce type constructors (s, or $\lfloor\tau\rfloor$) through shape checks. For every pre-value w stored on a heap $\mathcal{H}$, there is a corresponding entry in a blame map $\mathcal{B}$ that points to a set of boundaries. The blame map provides information if a mismatch occurs, following Reticulated Python (Vitousek et al., 2017; Vitousek, 2019).

Unlike for the higher-order-checking semantics, there is significant overlap between the Transient rules for typed and untyped redexes. Thus figure 18 presents one notion of reduction. The first group of rules in figure 18 handle boundary expressions and check expressions. When a value reaches a boundary, Transient matches its shape against the expected type. If successful, the value crosses the boundary and its blame map records the fact; otherwise, the program halts. For a dyn boundary, the result is a boundary error. For a stat boundary, the mismatch reflects an invariant error in typed code. Check expressions similarly match a value against a type-shape. On success, the blame map gains the boundaries associated with the location p_0 from which the value originated. On failure, these same boundaries may help the programmer diagnose the fault.

The second group of rules handle primitives and application. Pair projections and function applications must be followed by a check in typed contexts to enforce the type annotation at the elimination form. In untyped contexts, a check for the dynamic type embeds a possibly-typed subexpression. The binary operations are not elimination forms, so they are not followed by a check. Applications of typed functions additionally check the input value against the function's domain type. If successful, the blame map records the check. Otherwise, Transient reports the boundaries associated with the function (Vitousek et al., 2017). Note that untyped functions may appear in typed contexts, and vice-versa.

Applications of untyped functions in untyped code do not update the blame map. This allows an implementation to insert all checks by rewriting typed code at compile-time, leaving untyped code as is. Protected typed code can then interact with any untyped libraries.

Not shown in figure 18 are rules for elimination forms that halt the program. When δ is undefined or when a non-function is applied, the result is either an invariant error or a tag error depending on the context.

Transient shape checks do not guarantee full type soundness, complete monitoring, or the standard blame soundness and completeness. They do, however, preserve the top-level shape of all values in typed code. Furthermore, Transient satisfies a heap-based notion of blame soundness. Blame completeness fails because Transient does not update the blame map when an untyped function is applied in an untyped context.

Theorem 6.19. *Transient does not satisfy* **TS(1)**.

Proof Let $\mathsf{e}_0 = \mathsf{dyn}\ (\ell_0 \blacktriangleleft (\mathsf{Nat}\Rightarrow\mathsf{Nat}) \blacktriangleleft \ell_1)\ (\lambda x_0.\, -4)$.

$\boxed{\text{Transient Syntax}}$ extends Flat Evaluation Syntax

$v \;=\; i \mid n \mid p$

$\boxed{e; \mathcal{H}; \mathcal{B} \rhd_{T} e; \mathcal{H}; \mathcal{B}}$ selected rules

$$
\begin{array}{lll}
(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0); \mathcal{H}_0; \mathcal{B}_0 & \rhd_{T} & v_0; \mathcal{H}_0; (\mathcal{B}_0[v_0 \cup \{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}]) \\
\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0)) & & \\
(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0); \mathcal{H}_0; \mathcal{B}_0 & \rhd_{T} & \mathsf{BoundaryErr}\,(\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}, v_0); \mathcal{H}_0; \mathcal{B}_0 \\
\quad \text{if } \neg \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0)) & & \\
(\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0); \mathcal{H}_0; \mathcal{B}_0 & \rhd_{T} & v_0; \mathcal{H}_0; (\mathcal{B}_0[v_0 \cup \{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}]) \\
\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0)) & & \\
(\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0); \mathcal{H}_0; \mathcal{B}_0 & \rhd_{T} & \mathsf{InvariantErr}; \mathcal{H}_0; \mathcal{B}_0 \\
\quad \text{if } \neg \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0)) & & \\
(\mathsf{check}\{\mathcal{U}\}\, v_0\ p_0); \mathcal{H}_0; \mathcal{B}_0 & \rhd_{T} & v_0; \mathcal{H}_0; \mathcal{B}_0 \\
(\mathsf{check}\{\tau_0\}\, v_0\ p_0); \mathcal{H}_0; \mathcal{B}_0 & \rhd_{T} & v_0; \mathcal{H}_0; (\mathcal{B}_0[v_0 \cup \mathcal{B}_0(p_0)]) \\
\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0)) & & \\
(\mathsf{check}\{\tau_0\}\, v_0\ p_0); \mathcal{H}_0; \mathcal{B}_0 & \rhd_{T} & \mathsf{BoundaryErr}\,(\mathcal{B}_0(v_0) \cup \mathcal{B}_0(p_0), v_0); \mathcal{H}_0; \mathcal{B}_0 \\
\quad \text{if } \neg \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0)) & & \\
(\mathit{unop}\{{}^{\tau}/_{\mathcal{U}}\}\ p_0); \mathcal{H}_0; \mathcal{B}_0 & \rhd_{T} & (\mathsf{check}\{{}^{\tau}/_{\mathcal{U}}\}\ \delta(\mathit{unop}, \mathcal{H}_0(p_0))\ p_0); \mathcal{H}_0; \mathcal{B}_0 \\
\quad \text{if } \delta(\mathit{unop}, \mathcal{H}_0(p_0)) \text{ is defined} & & \\
(\mathit{binop}\{{}^{\tau}/_{\mathcal{U}}\}\ i_0\ i_1); \mathcal{H}_0; \mathcal{B}_0 & \rhd_{T} & \delta(\mathit{binop}, i_0, i_1); \mathcal{H}_0; \mathcal{B}_0 \\
\quad \text{if } \delta(\mathit{binop}, i_0, i_1) \text{ is defined} & & \\
(\mathsf{app}\{\tau_0\}\ p_0\ v_0); \mathcal{H}_0; \mathcal{B}_0 & \rhd_{T} & (\mathsf{check}\{\tau_0\}\ e_0[x_0 \leftarrow v_0]\ p_0); \mathcal{H}_0; \mathcal{B}_1 \\
\quad \text{if } \mathcal{H}_0(p_0) = \lambda x_0.\, e_0 & & \\
\quad \text{and } \mathcal{B}_1 = \mathcal{B}_0[v_0 \cup \mathit{rev}\,(\mathcal{B}_0(p_0))] & & \\
(\mathsf{app}\{\mathcal{U}\}\ p_0\ v_0); \mathcal{H}_0; \mathcal{B}_0 & \rhd_{T} & (e_0[x_0 \leftarrow v_0]); \mathcal{H}_0; \mathcal{B}_0 \\
\quad \text{if } \mathcal{H}_0(p_0) = \lambda x_0.\, e_0 & & \\
(\mathsf{app}\{{}^{\tau}/_{\mathcal{U}}\}\ p_0\ v_0); \mathcal{H}_0; \mathcal{B}_0 & \rhd_{T} & (\mathsf{check}\{{}^{\tau}/_{\mathcal{U}}\}\ e_0[x_0 \leftarrow v_0]\ p_0); \mathcal{H}_0; \mathcal{B}_1 \\
\quad \text{if } \mathcal{H}_0(p_0) = \lambda(x_0 : \tau_0).\, e_0 \text{ and } \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0)) & & \\
\quad \text{and } \mathcal{B}_1 = \mathcal{B}_0[v_0 \cup \mathit{rev}\,(\mathcal{B}_0(p_0))] & & \\
(\mathsf{app}\{{}^{\tau}/_{\mathcal{U}}\}\ p_0\ v_0); \mathcal{H}_0; \mathcal{B}_0 & \rhd_{T} & \mathsf{BoundaryErr}\,(\mathit{rev}\,(\mathcal{B}_0(p_0)), v_0); \mathcal{H}_0; \mathcal{B}_1 \\
\quad \text{if } \mathcal{H}_0(p_0) = \lambda(x_0 : \tau_0).\, e_0 \text{ and } \neg \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0)) & & \\
\quad \text{where } \mathcal{B}_1 = \mathcal{B}_0[v_0 \cup \mathit{rev}\,(\mathcal{B}_0(p_0))] & & \\
w_0; \mathcal{H}_0; \mathcal{B}_0 & \rhd_{T} & p_0; (\{p_0 \mapsto w_0\} \cup \mathcal{H}_0); (\{p_0 \mapsto \emptyset\} \cup \mathcal{B}_0) \\
\quad \text{where } p_0 \text{ fresh in } \mathcal{H}_0 \text{ and } \mathcal{B}_0 & &
\end{array}
$$

$\boxed{e; \mathcal{H}; \mathcal{B} \rightarrow^{*}_{T} e; \mathcal{H}; \mathcal{B}}$ $= \rightarrow^{*}_{T}$

where $E[e_0]; \mathcal{H}_0; \mathcal{B}_0 \;\; T \;\; E[e_1]; \mathcal{H}_1; \mathcal{B}_1$ if $e_0; \mathcal{H}_0; \mathcal{B}_0 \rhd_{T} e_1; \mathcal{H}_1; \mathcal{B}_1$

Fig. 18: Transient notions of reduction

- $\vdash e_0 : \mathsf{Nat}{\Rightarrow}\mathsf{Nat}$ in the surface syntax, but
- $e_0; \emptyset; \emptyset \rightarrow^*_{\mathsf{T}} p_0; \mathcal{H}_0; \mathcal{B}_0$, where $\mathcal{H}_0(p_0) = (\lambda x_0.\, {-4})$

and $\not\vdash_{\mathbf{1}} (\lambda x_0.\, {-4}) : \mathsf{Nat}{\Rightarrow}\mathsf{Nat}$. ■

Theorem 6.20. *Transient satisfies* **TS**(**s**).

Proof Recall that **s** maps types to type shapes and the unitype to itself. The proof depends on progress and preservation lemmas for the flat typing judgment ($\vdash_{\mathbf{s}}$). Although Transient lets any well-shaped value cross a boundary, the check expressions that appear after elimination forms preserve soundness. Suppose that an untyped function crosses a boundary and eventually computes an ill-typed result:

$$(\mathsf{app}\{\mathsf{Int}\}\, p_0\ 4); \mathcal{H}_0; \mathcal{B}_0 \ \rhd_{\mathsf{T}}\ (\mathsf{check}\{\mathsf{Int}\}\, \langle 4, \mathsf{sum}\{\mathcal{U}\}\, 4\ 1\rangle\ p_0); \mathcal{H}_0; \mathcal{B}_1$$
$$\text{if } \mathcal{H}_0(p_0) = \lambda x_0.\, \langle x_0, \mathsf{sum}\{\mathcal{U}\}\, x_0\ 1\rangle$$
$$\text{and } \mathcal{B}_1 = \mathcal{B}_0[v_0 \cup \mathit{rev}\,(\mathcal{B}_0(p_0))]$$

The check expression guards the context. ■

Theorem 6.21. *Transient does not satisfy* **CM**.

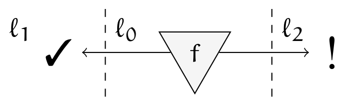
Proof A structured value can cross any boundary with a matching shape, regardless of the deeper type structure. For example, the following annotated rule adds a new label to a pair:

$$(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\ (\!(p_0)\!)^{\overline{\ell}_2})^{\ell_3}; \mathcal{H}_0; \mathcal{B}_0 \ \rhd_{\overline{\mathsf{T}}}\ (\!(p_0)\!)^{\overline{\ell}_2 \ell_3}; \mathcal{H}_0; \mathcal{B}_1$$
$$\text{where } \mathcal{H}_0(p_0) \in \langle v, v\rangle$$

■

Theorem 6.22. *Transient does not satisfy* **BS**.

Proof Let component ℓ_0 define a function f_0 and export it to components ℓ_1 and ℓ_2. If component ℓ_2 triggers a type mismatch, as sketched below, then Transient blames both component ℓ_2 and the irrelevant ℓ_1:



The following term expresses this scenario using a let-expression to abbreviate untyped function application:

$$(\mathsf{let}\ f_0 = (\lambda x_0.\, \langle x_0, x_0\rangle)\ \mathsf{in}$$
$$\mathsf{let}\ f_1 = (\mathsf{stat}\ (\ell_0 \blacktriangleleft (\mathsf{Int}{\Rightarrow}\mathsf{Int}) \blacktriangleleft \ell_1)\ (\mathsf{dyn}\ (\ell_1 \blacktriangleleft (\mathsf{Int}{\Rightarrow}\mathsf{Int}) \blacktriangleleft \ell_0)\ (f_0)^{\ell_0})^{\ell_1})\ \mathsf{in}$$
$$\mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_2)\ (\mathsf{app}\{\mathsf{Int}\}\, (\mathsf{dyn}\ (\ell_2 \blacktriangleleft (\mathsf{Int}{\Rightarrow}\mathsf{Int}) \blacktriangleleft \ell_0)\ (f_0)^{\ell_0})\ 5)^{\ell_2})^{\ell_0}; \emptyset; \emptyset$$

Reduction ends in a boundary error that blames three components. ■

Theorem 6.23. *Transient does not satisfy* **BC**.

Proof An untyped function application in untyped code does not update the blame map:

$$(\mathsf{app}\{\mathcal{U}\}\,\mathsf{p}_0\;\mathsf{v}_0);\mathcal{H}_0;\mathcal{B}_0 \;\;\rhd_{\mathsf{T}}\;\; (\mathsf{e}_0[\mathsf{x}_0\leftarrow\mathsf{v}_0]);\mathcal{H}_0;\mathcal{B}_0$$
$$\text{if } \mathcal{H}_0(\mathsf{p}_0) = \lambda\mathsf{x}_0.\,\mathsf{e}_0$$

Such applications lead to incomplete blame when the function has previously crossed a type boundary. To illustrate, the term below uses an untyped identity function f_1 to coerce the type of another function (f_0). After the coercion, an application leads to type mismatch.

$$\begin{aligned}
&(\mathsf{let}\ \mathsf{f}_0 = \mathsf{stat}\ (\ell_0\blacktriangleleft\tau_0\blacktriangleleft\ell_1)\ (\mathsf{dyn}\ (\ell_1\blacktriangleleft\tau_0\blacktriangleleft\ell_2)\ (\lambda\mathsf{x}_0.\,\mathsf{x}_0))\ \mathsf{in}\\
&\ \mathsf{let}\ \mathsf{f}_1 = \mathsf{stat}\ (\ell_0\blacktriangleleft(\tau_0\Rightarrow\tau_1)\blacktriangleleft\ell_3)\ (\mathsf{dyn}\ (\ell_3\blacktriangleleft(\tau_0\Rightarrow\tau_1)\blacktriangleleft\ell_4)\ (\lambda\mathsf{x}_1.\,\mathsf{x}_1))\ \mathsf{in}\\
&\ \mathsf{stat}\ (\ell_0\blacktriangleleft(\mathsf{Int}\times\mathsf{Int})\blacktriangleleft\ell_5)\\
&\qquad(\mathsf{app}\{\mathsf{Int}\times\mathsf{Int}\}\,(\mathsf{dyn}\ (\ell_5\blacktriangleleft\tau_1\blacktriangleleft\ell_0)\ (\mathsf{app}\{\mathcal{U}\}\,\mathsf{f}_1\ \mathsf{f}_0)^{\ell_0})\ 42)^{\ell_5})^{\ell_0};\emptyset;\emptyset
\end{aligned}$$

Reduction ends in a boundary error that does not report the crucial labels ℓ_3 and ℓ_4. ■

The results so far paint a negative picture of the wrapper-free Transient semantics. It fails **CM** and **BC** because it has no interposition mechanism to keep track of type implications for untyped code. Additionally, its heap-based approach to blame fails **BS** because the blame heap conflates different paths in a program. If several clients use the same library function and one client encounters a type mismatch, everyone gets blamed.

We have struggled to find a positive characterization of @—tname—'s blame behavior. Complete monitoring and blame completeness appear unattainable, even in a weakened form, because Transient has no control over untyped code. Blame soundness, however, is possible under a relaxed specification of ownership that adds one guideline to the "natural laws" from section 4.4.1:

8. If an address gains a label, then so does the associated pre-value on the heap.

$$\mathsf{fst}\ (\mathsf{p}_0)^{\ell_0};\mathcal{H}_0;\mathcal{B}_0;\mathcal{O}_0 \rightarrow (\mathsf{p}_1)^{\ell_0};\mathcal{H}_0;\mathcal{B}_0;\mathcal{O}_0[\mathsf{p}_1\cup\{\ell_0\}]$$
$$\textit{where } \mathcal{H}_0(\mathsf{p}_0) = \langle\mathsf{p}_1,\mathsf{p}_2\rangle$$

Law 3 propagates the outer label, which goes up to the ownership heap ($\mathcal{O}$).

Intuitively, the new specification pushes all ownership labels onto the heap. Rather than push to the value heap ($\mathcal{H}$) directly, though, the extended model of Transient in the supplement introduces a parallel store ($\mathcal{O}$) analogous to the blame heap.

Merging ownership labels on the heap is a non-compositional behavior. A programmer cannot reason about a local expression without thinking about how the rest of the codebase may introduce new owners. Because of this whole-program action, it is unclear whether the weakened notions of blame that follow are useful guarantees to strive for. Nevertheless, they help characterize Transient.

Definition 6.24 (heap-based blame soundness and blame completeness)**.** *For all well-formed* e_0 *such that* $\mathsf{e}_0;\emptyset;\emptyset;\emptyset \rightarrow^*_{\mathsf{X}} \mathsf{BoundaryErr}\ (\mathsf{b}^*_0,\mathsf{v}_0);\mathcal{H}_0;\mathcal{B}_0;\mathcal{O}_0$:

- X *satisfies* **BS-h** *iff senders* $(\mathsf{b}^*_0) \subseteq$ *owners* $(\mathsf{v}_0)\cup\mathcal{O}_0(\mathsf{v}_0)$
- X *satisfies* **BC-h** *iff senders* $(\mathsf{b}^*_0) \supseteq$ *owners* $(\mathsf{v}_0)\cup\mathcal{O}_0(\mathsf{v}_0)$.

Theorem 6.25. *Transient satisfies* **BS-h**.

$\boxed{\mathcal{O}; \mathcal{L}; \ell \Vdash_{h} e; \mathcal{B}}$ selected rules

$$\frac{\textit{senders}\,(\mathcal{B}_0(\mathsf{p}_0)) \subseteq \mathcal{O}_0(\mathsf{p}_0)}{\mathcal{O}_0; \mathcal{L}_0; \ell_0 \Vdash_{h} \mathsf{p}_0; \mathcal{B}_0} \qquad \frac{\mathcal{O}_0; \mathcal{L}_0; \ell_0 \Vdash_{h} e_0; \mathcal{B}_0 \qquad \textit{senders}\,(\mathcal{B}_0(\mathsf{p}_0)) \subseteq \mathcal{O}_0(\mathsf{p}_0)}{\mathcal{O}_0; \mathcal{L}_0; \ell_0 \Vdash_{h} \mathsf{check}\{\tau_0\}\, e_0\, \mathsf{p}_0; \mathcal{B}_0}$$

Fig. 19: Heap-based ownership consistency for Transient

Proof By a preservation lemma for the $\Vdash_{h}$ judgment sketched in figure 19 and fully-defined in the supplement. The judgment ensures that the blame map records a subset of the true owners on each heap-allocated value. One subtle case of the proof concerns function application, because the unlabeled rule appears to blame a typed function (at address p_0) for an unrelated incompatible value:

$$\begin{array}{l}(\mathsf{app}\{\tau/\mathcal{U}\}\, \mathsf{p}_0\, v_0); \mathcal{H}_0; \mathcal{B}_0 \;\rhd_{\mathsf{T}}\; \mathsf{BoundaryErr}\,(\textit{rev}\,(\mathcal{B}_0(\mathsf{p}_0)), v_0); \mathcal{H}_0; \mathcal{B}_1 \\ \quad \text{if } \mathcal{H}_0(\mathsf{p}_0) = \lambda(x_0 : \tau_0).\, e_0 \text{ and } \neg\textit{shape-match}\,(\lfloor \tau_0 \rfloor, \mathcal{H}_0(v_0)) \\ \quad \text{where } \mathcal{B}_1 = \mathcal{B}_0[v_0 \cup \textit{rev}\,(\mathcal{B}_0(\mathsf{p}_0))]\end{array}$$

But, the value is not unrelated because the shape check happens when this value meets the function's type annotation; that is, after the function receives the input value. By law 4, the correct labeling matches the following outline:

$$(\mathsf{app}\{\tau/\mathcal{U}\}\, (\!(\mathsf{p}_0)\!)^{\overline{\ell}_0}\, (\!(v_0)\!)^{\overline{\ell}_1})^{\ell_0}; \ldots \;\rhd_{\overline{\mathsf{T}}}\; (\mathsf{BoundaryErr}\,(\ldots, (\!(v_0)\!)^{\overline{\ell}_1 \ell_0 \textit{rev}\,(\overline{\ell}_0)}))^{\ell_0}; \ldots$$

Additionally blaming $\mathcal{B}_0(v_0)$ seems like a useful change to the original Transient semantics because it offers more information. Thanks to heap-based ownership, the technical justification is that adding these boundaries preserves the **BS-h** property. ■

Theorem 6.26. *Transient does not satisfy* **BC-h**.

Proof Blame completeness fails because Transient does not update the blame map during an untyped function application. Refer to the proof of theorem 6.23 for an example. ■

Theorem 6.27. $\mathsf{F} \lesssim \mathsf{T}$.

Proof Indirectly, via $\mathsf{T} \eqsim \mathsf{A}$ (theorem 6.31) and $\mathsf{F} \lesssim \mathsf{A}$ (theorem 6.33). ■

6.9 Amnesic and its Properties

The Amnesic semantics employs the same dynamic checks as Transient but offers path-based blame information. Whereas Transient indirectly tracks blame through heap addresses, Amnesic uses trace wrappers to keep boundaries alongside the value at hand.

Amnesic bears a strong resemblance to the Forgetful semantics. Both use guard wrappers in the same way, keeping a sticky "inner" wrapper around typed values and a temporary "outer" wrapper in typed contexts. There are two crucial differences:

- When Amnesic removes a guard wrapper, it saves the boundary specification in a trace wrapper. The number of boundaries in a trace can grow without bound, but the number of wrappers around a value is limited to three.

$\boxed{\text{Amnesic Syntax}}$ extends Higher-Order Evaluation Syntax

$v = i \mid n \mid \langle v, v\rangle \mid \lambda x.\, e \mid \lambda(x : \tau).\, e \mid \mathbb{G}\ (\ell \blacktriangleleft \tau \Rightarrow \tau \blacktriangleleft \ell)\ v \mid \mathbb{G}\ (\ell \blacktriangleleft \tau \times \tau \blacktriangleleft \ell)\ v \mid \mathbb{T}\ b^*\ v$

$\boxed{e \triangleright_{\mathsf{A}} e}$ selected rules

$\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 \quad \triangleright_{\mathsf{A}} \quad \mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0$
  if $\textit{shape-match}\ (\lfloor \tau_0 \rfloor, v_0)$
  and $\textit{rem-trace}\ (v_0) \in \langle v, v\rangle \cup (\lambda(x : \tau).\, e) \cup (\mathbb{G}\ b\ v)$

$\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 \quad \triangleright_{\mathsf{A}} \quad v_0$
  if $\textit{shape-match}\ (\lfloor \tau_0 \rfloor, v_0)$ and $\textit{rem-trace}\ (v_0) \in i$

$\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 \quad \triangleright_{\mathsf{A}} \quad \mathsf{BoundaryErr}\ (\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\} \cup b_0^*, v_0)$
  if $\neg\textit{shape-match}\ (\lfloor \tau_0 \rfloor, v_0)$ and $b_0^* = \textit{get-trace}\ (v_0)$

$\mathsf{fst}\{\tau_0\}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)\ v_0) \quad \triangleright_{\mathsf{A}} \quad \mathsf{dyn}\ b_0\ (\mathsf{fst}\{\mathcal{U}\}\, v_0)$
  where $b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$

$\mathsf{snd}\{\tau_0\}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)\ v_0) \quad \triangleright_{\mathsf{A}} \quad \mathsf{dyn}\ b_0\ (\mathsf{snd}\{\mathcal{U}\}\, v_0)$
  where $b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$

$\mathsf{app}\{\tau_0\}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_1 \Rightarrow \tau_2 \blacktriangleleft \ell_1)\ v_0)\ v_1 \quad \triangleright_{\mathsf{A}} \quad \mathsf{dyn}\ b_0\ (\mathsf{app}\{\mathcal{U}\}\, v_0\ (\mathsf{stat}\ b_1\ v_1))$
  where $b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$ and $b_1 = (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0)$

$\boxed{e \blacktriangleright_{\mathsf{A}} e}$ selected rules

$\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 \quad \blacktriangleright_{\mathsf{A}} \quad \mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0$
  if $\textit{shape-match}\ (\lfloor \tau_0 \rfloor, v_0)$ and $v_0 \in \langle v, v\rangle \cup (\lambda(x : \tau).\, e)$

$\mathsf{stat}\ b_0\ (\mathbb{G}\ b_1\ (\mathbb{T}_?\ b_0^*\ v_0)) \quad \blacktriangleright_{\mathsf{A}} \quad \mathsf{trace}\ (\{b_0, b_1\} \cup b_0^*)\ v_0$
  if $b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$ and $\textit{shape-match}\ (\lfloor \tau_0 \rfloor, v_0)$
  and $v_0 \in \langle v, v\rangle \cup (\lambda x.\, e) \cup (\mathbb{G}\ b\ (\lambda(x : \tau).\, e)) \cup (\mathbb{G}\ b\ \langle v, v\rangle)$

$\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ i_0 \quad \blacktriangleright_{\mathsf{A}} \quad i_0$
  if $\textit{shape-match}\ (\lfloor \tau_0 \rfloor, i_0)$

$\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 \quad \blacktriangleright_{\mathsf{A}} \quad \mathsf{InvariantErr}$
  if $\neg\textit{shape-match}\ (\lfloor \tau_0 \rfloor, v_0)$

$\mathsf{fst}\{\mathcal{U}\}\ (\mathbb{T}_?\ b_0^*\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\ v_0)) \quad \blacktriangleright_{\mathsf{A}} \quad \mathsf{trace}\ b_0^*\ (\mathsf{stat}\ b_0\ (\mathsf{fst}\{\tau_0\}\, v_0))$
  where $b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$

$\mathsf{snd}\{\mathcal{U}\}\ (\mathbb{T}_?\ b_0^*\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\ v_0)) \quad \blacktriangleright_{\mathsf{A}} \quad \mathsf{trace}\ b_0^*\ (\mathsf{stat}\ b_0\ (\mathsf{snd}\{\tau_1\}\, v_0))$
  where $b_0 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)$

$\mathsf{app}\{\mathcal{U}\}\ (\mathbb{T}_?\ b_0^*\ (\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0))\ v_1 \quad \blacktriangleright_{\mathsf{A}} \quad \mathsf{trace}\ b_0^*\ (\mathsf{stat}\ b_0\ (\mathsf{app}\{\tau_2\}\, v_0\ e_0))$
  where $\tau_0 = \tau_1 \Rightarrow \tau_2$ and $b_0 = (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1)$ and $b_1 = (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0)$
  and $e_0 = (\mathsf{dyn}\ b_1\ (\textit{add-trace}\ (\textit{rev}\ (b_0^*), v_1)))$

$\mathsf{trace}\ b_0^*\ v_0 \quad \blacktriangleright_{\mathsf{A}} \quad v_1$
  where $v_1 = \textit{add-trace}\ (b_0^*, v_0)$

$\boxed{e \rightarrow_{\mathsf{A}}^{*} e} \;=\; \rightarrow^{*}_{\cup\{\triangleright_{\mathsf{A}}, \blacktriangleright_{\mathsf{A}}, \blacktriangleright, \triangleright\}}$

Fig. 20: Amnesic notions of reduction

$$\textit{add-trace}\,(b_0^*, v_0) = \begin{cases} v_0 & \\ \quad \text{if } b_0^* = \emptyset & \\ \mathbb{T}\,(b_0^* \cup b_1^*)\ v_1 & \\ \quad \text{if } v_0 = \mathbb{T}\,b_1^*\,v_1 & \\ \mathbb{T}\,b_0^*\,v_0 & \\ \quad \text{if } v_0 \not\in \mathbb{T}\,b^*\,v \text{ and } b_0^* \neq \emptyset & \end{cases}$$

$$\textit{get-trace}\,(v_0) = \begin{cases} b_0^* & \text{if } v_0 = \mathbb{T}\,b_0^*\,v_1 \\ \emptyset & \text{if } v_0 \not\in \mathbb{T}\,b^*\,v \end{cases}$$

$$\textit{rem-trace}\,(v_0) = \begin{cases} v_1 & \text{if } v_0 = \mathbb{T}\,b_0^*\,v_1 \\ v_0 & \text{if } v_0 \not\in \mathbb{T}\,b^*\,v \end{cases}$$

$$(\mathbb{T}_?\,b_0^*\,v_0) = v_1 \iff \textit{rem-trace}\,(v_1) = v_0 \text{ and } \textit{get-trace}\,(v_1) = b_0^*$$

Fig. 21: Metafunctions for Amnesic

- At elimination forms, Amnesic checks only the context's type annotation. Suppose an untyped function enters typed code at one type and is later used at a supertype ($\mathsf{app}\{\mathsf{Int}\}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft (\mathsf{Nat}{\Rightarrow}\mathsf{Nat}) \blacktriangleleft \ell_1)\ \lambda x_0.\, -7)\ 2$). Amnesic runs this application without error but Forgetful raises a boundary error.

Thus, the following wrapped values can occur at run-time. Note that $(\mathbb{T}_?\,b^*\,e)$ is short for an expression that may or may not have a trace wrapper (figure 21).

$$\begin{array}{rcl} v_s & = & \mathbb{G}\,b\,(\mathbb{T}_?\,b^*\,\langle v, v\rangle) \\ & | & \mathbb{G}\,b\,(\mathbb{T}_?\,b^*\,\lambda x.\,e) \\ & | & \mathbb{G}\,b\,(\mathbb{T}_?\,b^*\,(\mathbb{G}\,b\,\langle v, v\rangle)) \\ & | & \mathbb{G}\,b\,(\mathbb{T}_?\,b^*\,(\mathbb{G}\,b\,\lambda(x:\tau).\,e)) \end{array} \qquad \begin{array}{rcl} v_d & = & \mathbb{T}\,b^*\,i \\ & | & \mathbb{T}\,b^*\,\langle v, v\rangle \\ & | & \mathbb{T}\,b^*\,\lambda x.\,e \\ & | & \mathbb{T}_?\,b^*\,(\mathbb{G}\,b\,\langle v, v\rangle) \\ & | & \mathbb{T}_?\,b^*\,(\mathbb{G}\,b\,\lambda(x:\tau).\,e) \end{array}$$

The elimination rules for guarded pairs show the clearest difference between checks in Amnesic and Forgetful. Amnesic ignores the type in the guard. Forgetful ignores the type annotation on the primitive operation.

Figure 20 defines three metafunctions and one abbreviation for trace wrappers. The metafunctions extend, retrieve, and remove the boundaries associated with a value. The abbreviation lets reduction rules accept optionally-traced values.

Amnesic satisfies full type soundness thanks to guard wrappers and fails complete monitoring because it drops wrappers. This is no surprise, since Amnesic creates and removes guard wrappers in the same manner as Forgetful. Unlike the Forgetful semantics, Amnesic uses trace wrappers to remember the boundaries that a value has crossed. This information leads to sound and complete blame messages.

Theorem 6.28. *Amnesic satisfies* $\mathbf{TS}(\mathbf{1})$.

Proof By progress and preservation lemmas for the higher-order typing judgment ($\vdash_{\mathbf{1}}$). Amnesic creates and drops wrappers in the same manner as Forgetful (theorem 6.14), so the only interesting proof cases concern elimination forms. For example, when Amnesic extracts an element from a guarded pair it ignores the type in the guard ($\tau_1 \times \tau_2$):

$$\mathsf{fst}\{\tau_0\}\,(\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_1 \times \tau_2 \blacktriangleleft \ell_1)\ v_0) \ \rhd_{\mathsf{A}}\ \mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ (\mathsf{fst}\{\mathcal{U}\}\,v_0)$$

The new boundary enforces the context's assumption (τ_0) instead, but we know that τ_0 is a supertype of τ_1 by the definition of the higher-order typing judgment. ■

$\boxed{\mathcal{L}; \ell \Vdash_p e}$ selected rules, extends $\mathcal{L}; \ell \Vdash e$

$$\frac{b_0^* = \{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1) \cdots (\ell_{n-1} \blacktriangleleft \tau_{n-1} \blacktriangleleft \ell_n)\} \qquad \mathcal{L}_0; \ell_n \Vdash_p v_0}{\mathcal{L}_0; \ell_0 \Vdash_p (\mathbb{T}\ b_0^*\ \llparenthesis v_0 \rrparenthesis^{\ell_n \cdots \ell_1})^{\ell_0}}$$

Fig. 22: Path-based ownership consistency for trace wrappers

Theorem 6.29. *Amnesic does not satisfy* **CM**.

Proof Removing a wrapper creates a value with more than one label:

$$(\mathsf{stat}\ (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\ \llparenthesis \mathbb{G}\ b_1\ \llparenthesis \mathbb{T}\ b_0^*\ \llparenthesis \lambda x_0.\, x_0 \rrparenthesis^{\bar{\ell}_2} \rrparenthesis^{\bar{\ell}_3} \rrparenthesis^{\bar{\ell}_4})^{\ell_5} \blacktriangleright_{\bar{A}}$$
$$\llparenthesis \mathsf{trace}\ (\{(\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1), b_1\} \cup b_0^*)\ \llparenthesis \lambda x_0.\, x_0 \rrparenthesis^{\bar{\ell}_2} \rrparenthesis^{\bar{\ell}_3 \bar{\ell}_4 \ell_5}$$

■

Theorem 6.30. *Amnesic satisfies* **BS** *and* **BC**.

Proof By progress and preservation lemmas for a path-based consistency judgment, $\Vdash_p$, that weakens single-owner consistency to allow multiple labels around a trace-wrapped value. Unlike the heap-based consistency for Transient, which requires an entirely new judgment, path-based consistency only replaces the rules for trace wrappers (shown in figure 22) and trace expressions. Now consider the guard-dropping rule:

$$(\mathsf{stat}\ (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\ \llparenthesis \mathbb{G}\ b_1\ \llparenthesis \mathbb{T}\ b_0^*\ \llparenthesis \lambda x_0.\, x_0 \rrparenthesis^{\bar{\ell}_2} \rrparenthesis^{\bar{\ell}_3} \rrparenthesis^{\bar{\ell}_4})^{\ell_5} \blacktriangleright_{\bar{A}}$$
$$\llparenthesis \mathsf{trace}\ (\{(\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1), b_1\} \cup b_0^*)\ \llparenthesis \lambda x_0.\, x_0 \rrparenthesis^{\bar{\ell}_2} \rrparenthesis^{\bar{\ell}_3 \bar{\ell}_4 \ell_5}$$

Path-consistency for the redex implies that $\bar{\ell}_3$ and $\bar{\ell}_4$ match the component names on the boundary b_1, and that the client side of b_1 matches the outer sender ℓ_1. Thus the new labels on the result match the sender names on the two new boundaries in the trace. ■

Theorem 6.31. $T \eqsim A$.

Proof By a stuttering simulation between Transient and Amnesic. Amnesic may take extra steps at an elimination form and to combine traces into one wrapper. Transient takes extra steps to place pre-values on the heap and to conservatively check the result of elimination forms. In fact, the two compute equivalent results up to wrappers and blame. ■

Remark 6.32. *Transient satisfies the spirit of* **TS**(**1**).

Proof Amnesic satisfies **TS**(**1**) (theorem 6.28) and the simulation used in theorem 6.31 shows that the only difference in behavior is that Amnesic adds superficial wrappers. ■

Theorem 6.33. $F \lesssim A$.

$\boxed{\text{Erasure Syntax}}$ extends Erased Evaluation Syntax

$v \;=\; i \mid n \mid \langle v, v\rangle \mid \lambda x.\, e \mid \lambda(x:\tau).\, e$

$\boxed{e \rhd_{\mathsf{E}} e}$

$$
\begin{array}{ll}
\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 & \rhd_{\mathsf{E}}\ v_0 \\
\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 & \rhd_{\mathsf{E}}\ v_0 \\
\mathit{unop}\{\tau_0\}\, v_0 & \rhd_{\mathsf{E}}\ \mathsf{BoundaryErr}\ (\emptyset, v_0) \\
\quad \text{if } \delta(\mathit{unop}, v_0) \text{ is undefined} & \\
\mathit{unop}\{\mathcal{U}\}\, v_0 & \rhd_{\mathsf{E}}\ \mathsf{TagErr} \\
\quad \text{if } \delta(\mathit{unop}, v_0) \text{ is undefined} & \\
\mathit{unop}\{{}^{\tau}\!/\!_{\mathcal{U}}\}\, v_0 & \rhd_{\mathsf{E}}\ \delta(\mathit{unop}, v_0) \\
\quad \text{if } \delta(\mathit{unop}, v_0) \text{ is defined} & \\
\mathit{binop}\{\tau_0\}\, v_0\ v_1 & \rhd_{\mathsf{E}}\ \mathsf{BoundaryErr}\ (\emptyset, v_i) \\
\quad \text{if } \delta(\mathit{binop}, v_0, v_1) \text{ is undefined and } v_i \notin \mathsf{Int} & \\
\mathit{binop}\{\mathcal{U}\}\, v_0\ v_1 & \rhd_{\mathsf{E}}\ \mathsf{TagErr} \\
\quad \text{if } \delta(\mathit{binop}, v_0, v_1) \text{ is undefined} & \\
\mathit{binop}\{{}^{\tau}\!/\!_{\mathcal{U}}\}\, v_0\ v_1 & \rhd_{\mathsf{E}}\ \delta(\mathit{binop}, v_0, v_1) \\
\quad \text{if } \delta(\mathit{binop}, v_0, v_1) \text{ is defined} & \\
\mathsf{app}\{\tau_0\}\, v_0\ v_1 & \rhd_{\mathsf{E}}\ \mathsf{BoundaryErr}\ (\emptyset, v_0) \\
\quad \text{if } v_0 \notin (\lambda x.\, e) \cup (\lambda(x:\tau).\, e) & \\
\mathsf{app}\{\mathcal{U}\}\, v_0\ v_1 & \rhd_{\mathsf{E}}\ \mathsf{TagErr} \\
\quad \text{if } v_0 \notin (\lambda x.\, e) \cup (\lambda(x:\tau).\, e) & \\
\mathsf{app}\{{}^{\tau}\!/\!_{\mathcal{U}}\}\ (\lambda(x_0:\tau_0).\, e_0)\ v_0 & \rhd_{\mathsf{E}}\ e_0[x_0 \leftarrow v_0] \\
\mathsf{app}\{{}^{\tau}\!/\!_{\mathcal{U}}\}\ (\lambda x_0.\, e_0)\ v_0 & \rhd_{\mathsf{E}}\ e_0[x_0 \leftarrow v_0]
\end{array}
$$

Fig. 23: Erasure notions of reduction

Proof By a lock-step bisimulation. The only difference between Forgetful and Amnesic comes from subtyping. Forgetful uses wrappers to enforce the type on a boundary. Amnesic uses boundary types only for an initial shape check, and instead uses the static types in typed code to guide checks at elimination forms. In the following $\mathsf{A} \not\lesssim \mathsf{F}$ example, a boundary declares one type and an elimination form requires a weaker type:

$$\mathsf{fst}\{\mathsf{Int}\}\ (\mathsf{dyn}\ (\ell_0 \blacktriangleleft (\mathsf{Nat}{\times}\mathsf{Nat}) \blacktriangleleft \ell_1)\ \langle -4, 4\rangle)$$

Since -4 is an integer, Amnesic reduces to a value. Forgetful detects an error. ■

6.10 *Erasure and its Properties*

Figure 23 presents the values and notions of reduction for the Erasure semantics. Erasure ignores all types at runtime. As the first two reduction rules show, any value may cross any boundary. When an incompatible value reaches an elimination form, the result depends on the context. In untyped code, the redex steps to a standard tag error. In typed code, however, the malformed redex indicates that an ill-typed value crossed a boundary. Thus

Erasure ends with a boundary error at the last possible moment; these errors come with no information because there is no record of the relevant boundary.

Theorem 6.34. *Erasure satisfies neither* $\mathbf{TS}(\mathbf{1})$ *nor* $\mathbf{TS}(\mathbf{s})$.

Proof Dynamic-to-static boundaries are unsound. An untyped function, for example, can enter a typed context that expects an integer:

$$\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_1)\ (\lambda x_0.\,42)\ \rhd_{\mathsf{E}}\ (\lambda x_0.\,42)$$

■

Theorem 6.35. *Erasure satisfies* $\mathbf{TS}(\mathbf{0})$.

Proof By progress and preservation lemmas for the erased "dynamic-typing" judgment ($\vdash_{\mathbf{0}}$). Given well-formed input, every $\rhd_{\mathsf{E}}$ rule yields a dynamically-typed result. ■

Theorem 6.36. *Erasure does not satisfy* **CM**.

Proof A static-to-dynamic boundary can create a value with multiple labels:

$$(\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ (v_0)^{\ell_2})^{\ell_3}\ \rhd_{\overline{\mathsf{E}}}\ ((v_0))^{\ell_2\ell_3}$$

■

Theorem 6.37.

- *Erasure satisfies* **BS**.
- *Erasure does not satisfy* **BC**.

Proof An Erasure boundary error blames an empty set, for example:

$$\mathsf{fst}\{\mathsf{Int}\}\ (\lambda x_0.\,x_0)\ \rhd_{\mathsf{E}}\ \mathsf{BoundaryErr}\,(\emptyset,\ (\lambda x_0.\,x_0))$$

The empty set is trivially sound and incomplete. ■

Theorem 6.38. $\mathsf{A} \lesssim \mathsf{E}$.

Proof By a stuttering simulation. Amnesic takes extra steps at elimination forms, to enforce types, and to create trace wrappers.

As a counterexample showing $\mathsf{A} \not\gtrsim \mathsf{E}$, the following applies an untyped function:

$$\mathsf{app}\{\mathsf{Nat}\}\ (\mathsf{dyn}\ (\ell_0 \blacktriangleleft (\mathsf{Nat}{\Rightarrow}\mathsf{Nat}) \blacktriangleleft \ell_1)\ (\lambda x_0.\,{-9}))\ 4$$

Amnesic checks for a natural-number result and errors, but Erasure checks nothing. ■

7 Discussion

One central design issue of gradual typing is the semantics of types and specifically how their integrity is enforced at the boundaries between typed and untyped code. Among other things, the choice determines whether typed code can trust the static types and the quality of assistance that a programmer receives when a mixed-typed interaction goes wrong.

Without an interaction story, mixed-typed programs are no better than dynamically-typed programs when it comes to run-time errors. Properties that hold for the typed half of the language are only valid under a closed-world assumption (Bierman et al., 2014; Rastogi et al., 2015; Chaudhuri et al., 2017); such properties are an important starting point, but make no contribution to the overall goal.

As the analysis of this paper demonstrates, the limitations of the host language determine the invariants that a language designer can hope to enforce. First, higher-order wrappers enable strong guarantees, but functional APIs or support from the host runtime system. A language without wrappers of any sort can provide weaker guarantees by rewriting typed code and maintaining a global map of blame metadata. If this metadata can be attached directly to a value, then stronger blame guarantees are in reach.

More precisely, this paper analyzes six distinct semantics via four properties (table 2) and establishes an error preorder relation:

- Type soundness is a relatively weak property for mixed-typed programs; it determines whether typed code can trust its own types. Except for the Erasure semantics, which does nothing to enforce types, type soundness does not clearly distinguish the various strategies (remark 6.32).
- Complete monitoring is a stronger property, adapted from the literature on higher-order contracts (Dimoulas et al., 2012). It holds when *untyped* code can trust type specifications and vice-versa; see section 2.3 for examples.

The last two properties tell a developer what aid to expect if a type mismatch occurs.

- Blame soundness states that every boundary in a blame message is potentially responsible. Four strategies satisfy blame soundness relative to a path-based notion of responsibility. Transient satisfies blame soundness only if the notion of responsibility is weakened to merge distinct references to the same heap-allocated value. Erasure is trivially blame-sound because it gives the programmer zero information.
- Blame completeness states that every blame error comes with an overapproximation of the responsible parties. Three of the four blame-sound semantics also satisfy blame completeness. Forgetful can satisfy this property. The Erasure strategy trivially fails blame completeness. And the Transient strategy fails because it has no way to supervise untyped values that flow through a typed context.

Table 2 notes, however, that the weakest strategies are the only ones that do not require wrapper values. Wrappers impose space costs, time costs, and object identity issues (Strickland et al., 2012; Vitousek et al., 2014; Keil et al., 2015), but seem essential for strong mixed-typed guarantees. Perhaps future work can find a way to satisfy additional properties without using wrappers.

The design of a semantics of type enforcement has implications for two other, major aspects of the design of a gradually typed language: the performance of its implementation and its acceptance by working developers. Greenman et al. (2019b) developed an evaluation framework for the performance concern that is slowly gaining in acceptance, Tunnell Wilson et al. (2018) present rather preliminary results concerning the acceptance by programmers. In conclusion, though, all three problem areas are barely understood.

Table 2: Technical contributions

	N	C	F	T	A	E
type soundness	**1**	**1**	**1**	$\mathbf{s}^{\dagger}$	**1**	**0**
complete monitoring	✓	✓	×	×	×	×
blame soundness	✓	✓	✓	*heap*	✓	∅
blame completeness	✓	✓	$\times^{\ddagger}$	×	✓	×
error preorder	N $\lesssim$	C $\lesssim$	F $\lesssim$	T $\approx$	A $\lesssim$	E
no wrappers	×	×	×	✓	×	✓

† indirectly satisfies **TS**(**1**) by a bisimulation to A (theorem 6.31)
‡ satisfiable by adding A-style trace wrappers, see supplement

Much remains to be done before the community can truly claim to understand this complex design space.

Acknowledgments

Michael Ballantyne inspired section 5 and pushed us towards finer comparisons. Michael M. Vitousek suggested that Transient is not as unsound as it first seems (remark 6.32). Amal Ahmed, Stephen Chang, and Max S. New criticized several explanations of complete monitoring. NSF grants `CCF 1518844` and `CCF 1763922` provided support.

References

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *POPL*, pages 201–214, 2011.

Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, 96(1):52–69, 2013.

Deyaaeldeen Almahallawi. *Towards Efficient Gradual Typing via Monotonic References and Coercions*. PhD thesis, Indiana University, 2020.

Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. Sound gradual typing: only mostly dead. *PACMPL*, 1(OOPSLA):54:1–54:24, 2017.

Jan A. Bergstra and John V. Tucker. Initial and final algebra semantics for data type specifications: Two characterization theorems. *SIAM J. Comput.*, 12(2):366–387, 1983.

Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C#. In *ECOOP*, pages 76–100, 2010.

Gavin Bierman, Martin Abadi, and Mads Torgersen. Understanding TypeScript. In *ECOOP*, pages 257–281, 2014.

Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: Robust, concurrent, extensible scripting on the JVM. In *OOPSLA*, 2009.

Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical optional types for Clojure. In *ESOP*, pages 68–94, 2016.

Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA*, pages 215–230, 1993.

Robert Cartwright. A constructive alternative to data type definitions. In *LISP and functional programming*, pages 46–55, 1980.

Giuseppe Castagna and Victor Lanvin. Gradual typing with union and intersection types. *PACMPL*, 1(ICFP):41:1–41:28, 2017.

Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levy. Fast and precise type checking for javascript. *PACMPL*, 1(OOPSLA):56:1–56:30, 2017.

Olaf Chitil. Practical typed lazy contracts. In *ICFP*, pages 67–76, 2012.

Benjamin W. Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. KafKa: Gradual typing for objects. In *ECOOP*, pages 12:1–12:23, 2018.

Dart. The Dart type system, 2020. URL `https://dart.dev/guides/language/type-system`. Accessed 2020-09-04.

Markus Degen, Peter Thiemann, and Stefan Wehr. The interaction of contracts and laziness. In *PEPM*, pages 97–106, 2012.

Christos Dimoulas and Matthias Felleisen. On contract satisfaction in a higher-order world. *TOPLAS*, 33(5):16:1–16:29, 2011.

Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: no more scapegoating. In *POPL*, pages 215–226, 2011.

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *ESOP*, pages 214–233, 2012.

Christos Dimoulas, Max S. New, Robert Bruce Findler, and Matthias Felleisen. Oh lord, please don't let contracts be misunderstood (functional pearl). In *ICFP*, pages 117–131, 2016.

Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. Collapsible contracts: Fixing a pathology of gradual typing. *PACMPL*, 2(OOPSLA): 133:1–133:27, 2018.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP*, pages 48–59, 2002.

Robert Bruce Findler, Shu yu Guo, and Anne Rogers. Lazy contract checking for immutable data structures. In *IFL*, pages 111–128, 2007.

Michael Greenberg. Space-efficient manifest contracts. In *POPL*, pages 181–194, 2015.

Michael Greenberg. The dynamic practice and static theory of gradual typing. In *SNAPL*, pages 6:1–6:20, 2019.

Ben Greenman and Matthias Felleisen. A spectrum of type soundness and performance. *PACMPL*, 2(ICFP):71:1–71:32, 2018.

Ben Greenman and Zeina Migeed. On the cost of type-tag soundness. In *PEPM*, pages 30–39, 2018.

Ben Greenman, Matthias Felleisen, and Christos Dimoulas. Complete monitors for gradual types. *PACMPL*, 3(OOPSLA):122:1–122:29, 2019a.

Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. How to evaluate the performance of gradual type systems. *JFP*, 29(e4):1–45, 2019b.

Hugo Musso Gualandi and Roberto Ierusalimschy. Pallene: a companion language for Lua. *Science of Computer Programming*, 189(102393):1–15, 2020.

David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *HOSC*, 23(2): 167–189, 2010.

Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *FLOPS*, pages 208–225, 2006.

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. On polymorphic gradual typing. *PACMPL*, 1 (ICFP):40:1–40:29, 2017.

Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. Transparent object proxies in JavaScript. In *ECOOP*, pages 149–173, 2015.

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. Toward efficient gradual typing for structural types via coercions. In *PLDI*, pages 517–532, 2019.

Andre Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. A formalization of Typed Lua. In *DLS*, pages 13–25, 2015.

Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *TOPLAS*, 31(3):1–44, 2009.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

David A. Moon. MACLISP reference manual, revision 0. Technical report, MIT Project MAC, 1974.

Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. Extensible access control with authorization contracts. In *OOPSLA*, pages 214–233, 2016.

Fabian Muehlboeck and Ross Tate. Sound gradual typing is nominally alive and well. *PACMPL*, 1 (OOPSLA):56:1–56:30, 2017.

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

Max S. New, Dustin Jamner, and Amal Ahmed. Graduality and parametricity: together again for the first time. *PACMPL*, 4(POPL):46:1–46:32, 2020.

Atsushi Ohori and Kazuhiko Kato. Semantics for communication primitives in a polymorphic language. In *POPL*, pages 99–112, 1993.

Norman Ramsey. Embedding an interpreted language using higher-order functions and types. *JFP*, 21(6):585–615, 2008.

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. In *POPL*, pages 167–180, 2015.

Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. The ruby type checker. In *SAC*, pages 1565–1572, 2013.

Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete types for TypeScript. In *ECOOP*, pages 76–100, 2015.

Gregor Richards, Ellen Arteca, and Alexi Turcotte. The vm already knew that: Leveraging compile-time knowledge to optimize gradual typing. *PACMPL*, 1(OOPSLA):55:1–55:27, 2017.

Richard Roberts, Stefan Marr, Michael Homer, and James Noble. Transient typechecks are (almost) free. In *ECOOP*, pages 15:1–15:29, 2019.

Jeremy Siek, Peter Thiemann, and Philip Wadler. Blame and coercion: Together again for the first time. In *PLDI*, pages 425–435, 2015a.

Jeremy Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic references for efficient gradual typing. In *ESOP*, pages 432–456, 2015b.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming. University of Chicago, TR-2006-06*, 2006.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *SNAPL*, pages 274–293, 2015c.

Guy L. Steele, Jr. *Common Lisp*. Digital Press, 2nd edition, 1990.

T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: Run-time support for reasonable interposition. In *OOPSLA*, pages 943–962, 2012.

Nikhil Swamy, Cédric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in JavaScript. In *POPL*, pages 425–437, 2014.

Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *OOPSLA*, pages 793–810, 2012.

Asumu Takikawa, Daniel Feltey, Earl Dean, Robert Bruce Findler, Matthew Flatt, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards practical gradual typing. In *ECOOP*, pages 4–27, 2015.

Satish Thatte. Quasi-static typing. In *POPL*, pages 367–381, 1990.

Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *DLS*, pages 964–974, 2006.

Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *POPL*, pages 395–406, 2008.

Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *ICFP*, pages 117–128, 2010.

Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory typing: ten years later. In *SNAPL*, pages 17:1–17:17, 2017.

Matías Toro, Elizabeth Labrada, and Éric Tanter. Gradual parametricty, revisited. *PACMPL*, 3 (POPL):17:1–17:30, 2019.

Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. The behavior of gradual types: a user study. In *DLS*, pages 1–12, 2018.

Michael M. Vitousek. *Gradual Typing for Python, Unguarded*. PhD thesis, Indiana University, 2019.

Michael M. Vitousek, Andrew Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for python. In *DLS*, pages 45–56, 2014.

Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big types in little runtime: open-world soundness and collaborative blame for gradual type systems. In *POPL*, pages 762–774, 2017.

Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. Optimizing and evaluating transient gradual typing. In *DLS*, pages 28–41, 2019.

Philip Wadler. A complement to blame. In *SNAPL*, pages 309–320, 2015.

Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *ESOP*, pages 1–15, 2009.

Mitchell Wand. Final algebra semantics and data type extensions. *Journal of Computer and System Sciences*, 19:27–44, 1979.

Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. Mixed messages: Measuring conformance and non-interference in TypeScript. In *ECOOP*, pages 28:1–28:29, 2017.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, pages 38–94, 1994. First appeared as Technical Report TR160, Rice University, 1991.

Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *POPL*, pages 377–388, 2010.