Typed–Untyped Interactions: A Comparative Analysis

BEN GREENMAN*, PLT @ Brown University, USA
CHRISTOS DIMOULAS, PLT @ Northwestern University, USA
MATTHIAS FELLEISEN, PLT @ Northeastern University, USA

The literature presents many strategies for enforcing the integrity of types when typed code interacts with untyped code. This paper presents a uniform evaluation framework that characterizes the differences among some major existing semantics for typed–untyped interaction. Type system designers can use this framework to analyze the guarantees of their own dynamic semantics.

Additional Key Words and Phrases: complete monitoring, blame soundness, blame completeness

ACM Reference Format:
Ben Greenman, Christos Dimoulas, and Matthias Felleisen. 2023. Typed–Untyped Interactions: A Comparative Analysis. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (January 2023), 55 pages. https://doi.org/10.1145/3579833

1 CALLING ALL TYPES

Many programming languages let typed code interact with untyped code in some ways while retaining some desirable aspects of each typing discipline. The currently popular research focus of gradual typing provides many examples. Exactly which interactions are allowed and which desirable aspects are retained, however, varies widely among languages. There are four leading *type-enforcement strategies* that restrict interactions between typed and untyped code:

- *Erasure* (aka. optional typing) is a hands-off method that uses types only for static analysis and imposes no restrictions at run-time [8, 11].
- *Transient* inserts shape checks[1] in typed code to guarantee only that operations cannot "go wrong" in the *typed portion* of code due to values from the untyped portion [83, 86].
- *Natural* uses higher-order checks to ensure the integrity of types in the *entire program* [68, 78].
- *Concrete* enforces types with tag checks. It ensures the full integrity of types, but requires that every value comes with a fully descriptive type tag [52, 93].

In addition, researchers have designed hybrid techniques [9, 31, 34, 61, 64]. An outstanding and unusual exemplar of this kind is Pyret, a language targeting the educational realm (`pyret.org`).

Each semantic choice denotes a trade-off among static guarantees, expressiveness, and run-time costs. Language designers should understand these trade-offs when they create a new typed–untyped interface. Programmers need to appreciate the trade-offs if they can choose a language for

---

*Research completed at Northeastern University prior to joining Brown

[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.

Authors' addresses: Ben Greenman, PLT @ Brown University, Providence, Rhode Island, USA, benjaminlgreenman@gmail.com; Christos Dimoulas, PLT @ Northwestern University, Evanston, Illinois, USA, chrdimo@northwestern.edu; Matthias Felleisen, PLT @ Northeastern University, Boston, Massachusetts, USA, matthias@ccs.neu.edu.

0164-0925/2023/1-ART1 $15.00
https://doi.org/10.1145/3579833

Table 1. Informal sketch of contributions; full results in table 2 (page 52).

	Natural		Co-Natural		Forgetful		Transient		Amnesic		Erasure
type soundness	✓		✓		✓		✓		✓		×
complete monitoring	✓		✓		×		×		×		×
blame soundness	✓		✓		✓		×		✓		✓
blame completeness	✓		✓		×		×		✓		×
error preorder	N	$\lesssim$	C	$\lesssim$	F	$\lesssim$	T	$\eqsim$	A	$\lesssim$	E
no wrappers	×		×		×		✓		×		✓

a project. 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 proxy values, then Transient may be the most desirable option. If fine-grained interoperability demands complete type integrity everywhere, Natural is the right choice.[2] And if predictable behavior and performance matters most, then Erasure may be best—it is certainly the industry favorite.

Unfortunately, the literature provides little guidance about how to compare such different semantics formally. For example, the dynamic gradual guarantee [69]—a widely studied property in the gradual typing world—is satisfied by any type-enforcement strategy, including the no-check Erasure, as long as the type `Dynamic` is relatively well-behaved.[3] In short, the field lacks an apples-to-apples way of comparing different strategies and considering their implications.

This paper introduces a framework for systematically comparing the behavioral guarantees offered by different semantics of typed–untyped interaction. The comparison begins with a common surface syntax to express programs that can mix typed and untyped code. This surface syntax is then assigned multiple semantics, each of which follows a distinct protocol for enforcing the integrity of types across boundaries. With this framework, one can directly study the possible behaviors for a single program.

Using the framework, the paper compares the three implemented semantics explained above (Natural (N), Transient (T), Erasure (E)) and three theoretical ones (Co-Natural (C), Forgetful (F), and Amnesic (A)). *Co-Natural* enforces data structures lazily rather than eagerly. *Forgetful* is lazy in the same way and also ignores type obligations that are not strictly required for type soundness. *Amnesic* is a variation of Transient that uses wrappers to improve its blame guarantees.

The comparison excludes two classes of prior work: Concrete, because of the stringent constraints it places on untyped code, and semantics that rely on an analysis of the untyped code (such as [2, 13, 91]). That is, the focus is on enforcement strategies that can deal with untyped code as a "dusty deck" without recompiling the untyped world each time a new type boundary appears.

Table 1 sketches the results of the evaluation. The six letters in the top row correspond to 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—a property adapted from research on contracts [23]—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 contains *only* boundaries that are relevant to the problem. Blame completeness guarantees that error messages

---

[2] Implementations of Natural can yield performance improvements relative to untyped code, especially when typed code rarely interacts with untyped code [44, 75].

[3] Thanks to the TOPLAS reviewers for reminding us that the gradual guarantees are not meant to distinguish semantics in terms of how they enforce types. The guarantees address a separate dimension; namely, the behavior of type `Dynamic`.

come with *all* relevant information, though possibly with some irrelevant extras. For both blame soundness and completeness, the notion of relevant boundaries is determined by an independent (axiomatic) specification that tracks values as they cross boundaries between typed and untyped code. Lastly, the error preorder compares the relative permissiveness of types in two semantics. Natural (N) accepts the fewest programs without raising a run-time type mismatch and Erasure (E) accepts the greatest number of programs. Additionally, Transient and Erasure are the only strategies that can avoid the complexity of wrapper values.

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) is the result of a search for a semantics that fails complete monitoring but guarantees sound and complete blame.

1.1 Performance and Pragmatics are Out of Scope

Understanding the formal properties of typed–untyped interactions is only one third of the challenge. Two parallel and ongoing quests aim to uncover the performance implications of different strategies [6, 24, 34, 37, 38, 44] and the pragmatics of the semantics for working developers [45]. These efforts fall outside the scope of this paper.

1.2 Relation to Prior Work

This paper is a synthesis of results that have been published piecemeal in two conference papers [34, 35] and a dissertation chapter [33]. It is the only paper to compare the six semantics on equal grounds. In addition to the synthesis, it brings three contributions: a survey of type-enforcement strategies, a high-level comparison of the six semantics, and refined meta-theoretic results.

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 languages that support typed–untyped interactions that needs organizing principles (section 2). The properties listed in the top five rows of table 1 offer an expressive and scalable basis for comparison (section 3). By starting with a common surface language and defining semantics that explore various strategies for enforcing types, the properties enable apples-to-apples comparisons of the dynamics of typed–untyped interactions (section 4). This paper focuses on six type-enforcement strategies in particular (section 5).

Section 6 formally presents the six semantics and the key results. Expert readers who are not interested in informal discussions may wish to begin there and use section 5 as needed for a high-level picture. The supplementary material presents the essential definitions, lemmas, and proof sketches that support the results.

2 ASSORTED BEHAVIORS BY EXAMPLE

There are many languages that allow typed and untyped code to interact. Figure 1 arranges a few of their names into a rough picture of the design space. Languages marked with a star (★) are *gradual* in the sense that they come with a universal dynamic type, often styled as `Dynamic`, ★, or ? [68, 77]. Technically, the type system supports implicit down-casts from the dynamic type to any other type—unlike, say, the universal `Object` type in Java. This notion of gradual is more permissive than the refined one from Siek et al. [69], which asks for a dynamic type that satisfies the gradual guarantees [69]. Languages marked with a cross (†) are *migratory* [81]; they add a tailor-made type system to an untyped language (as opposed to working static-first [32]). Other languages have different priorities. This paper uses the name "mixed-typed" as an umbrella term to describe languages in the design space.

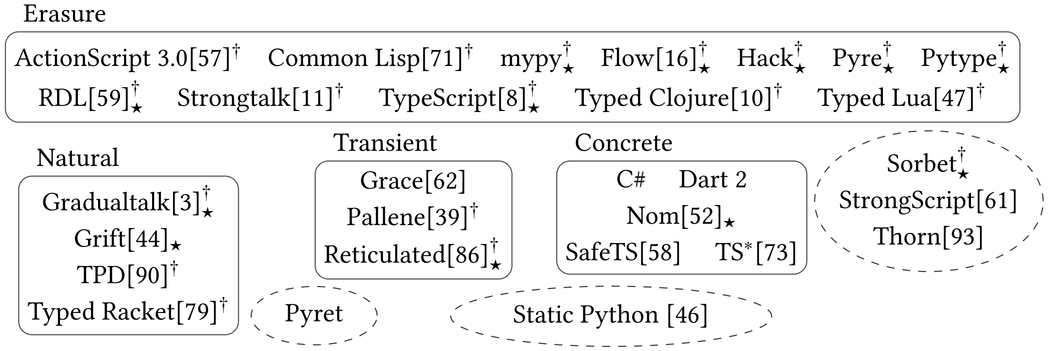


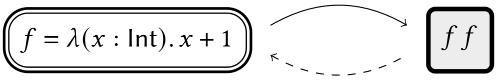
Fig. 1. Landscape of mixed-typed languages, † = migratory, ★ = gradual

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. The Natural languages come from academic teams that are interested in types that offer strong guarantees, Transient is gaining attention as a compromise between types and performance, and Concrete has generated interest among industry teams as well as academics. Several languages exhibit a hybrid approach. Sorbet adds types to Ruby and optionally checks method signatures at run-time. Thorn and StrongScript offer both concrete and erased types [61, 93]. Pyret uses Natural-style checks to validate fixed-size data and Transient-style checks for recursive types (e.g. lists) and higher-order types.[4] Static Python combines Transient and Co-Natural to mitigate the restrictions of the latter [46]. Grift has a second mode that implements a monotonic semantics [4]. Prior to its 2.0 release, Dart took a hybrid approach. Developers could toggle between a checked mode and an Erasure mode. Monotonic is similar to Natural, but uses a checked heap instead of wrappers and rejects additional programs [58, 60, 64, 73]. A final variant is from the literature. Castagna and Lanvin [15] present a semantics that creates wrappers like Natural but also removes wrapper that do not matter for type soundness. This semantics is similar to the forgetful contract semantics [31].

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 statically-typed code and dynamically-typed code. The following three subsections present illustrative examples of interactions between typed and untyped code in four programming languages: Flow [16], Reticulated [86], Typed Racket [81], and Nom [52]. These languages use the Erasure, Transient, Natural, and Concrete strategies, respectively. 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.

2.1 Enforcing a Base Type

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



(1)

---

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

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

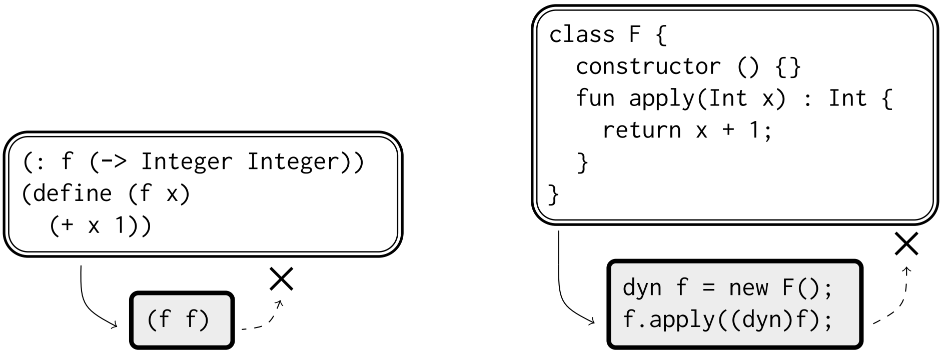
```
f(f);
```

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

```
f(f)
```

(a) Flow

(b) Reticulated



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

```
(f f)
```

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

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

(c) Typed Racket

(d) Nom

Fig. 2. Program (1) translated to four languages

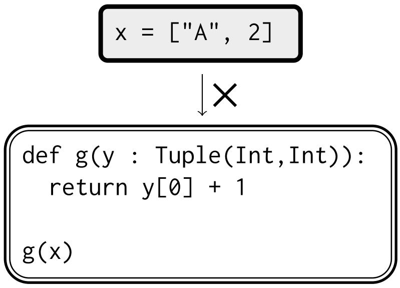
The typed function on the left expects an integer. The untyped context on the right imports this function *f* and applies *f* to itself; thus the typed function receives a function rather than an integer. The question is whether the program halts or invokes the typed function *f* on a nonsensical input.

Figure 2 translates the program to the four chosen languages. Each white box represents type-checked code, and each grey box represents untyped and un-analyzed code. The arrows represent the boundary behavior: the solid arrow stands for the call from one area to the other, and the dashed one for the return. 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 and applies the function to itself in an untyped context.

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

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.

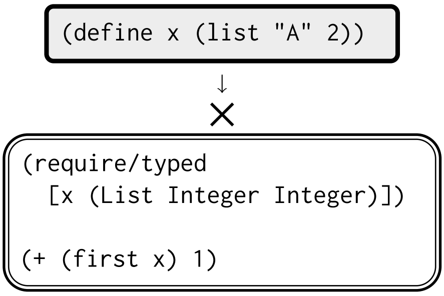


```
x = ["A", 2]
```

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

g(x)
```

(a) Reticulated

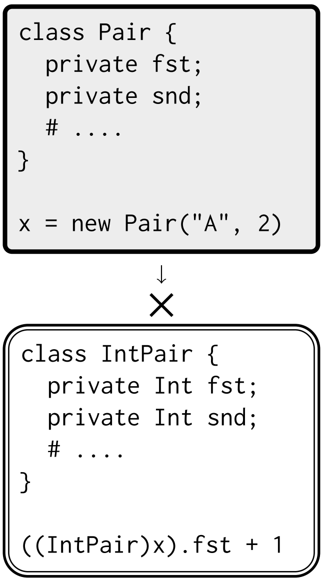


```
(define x (list "A" 2))
```

```
(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) translations

2.2 Validating an Untyped Data Structure

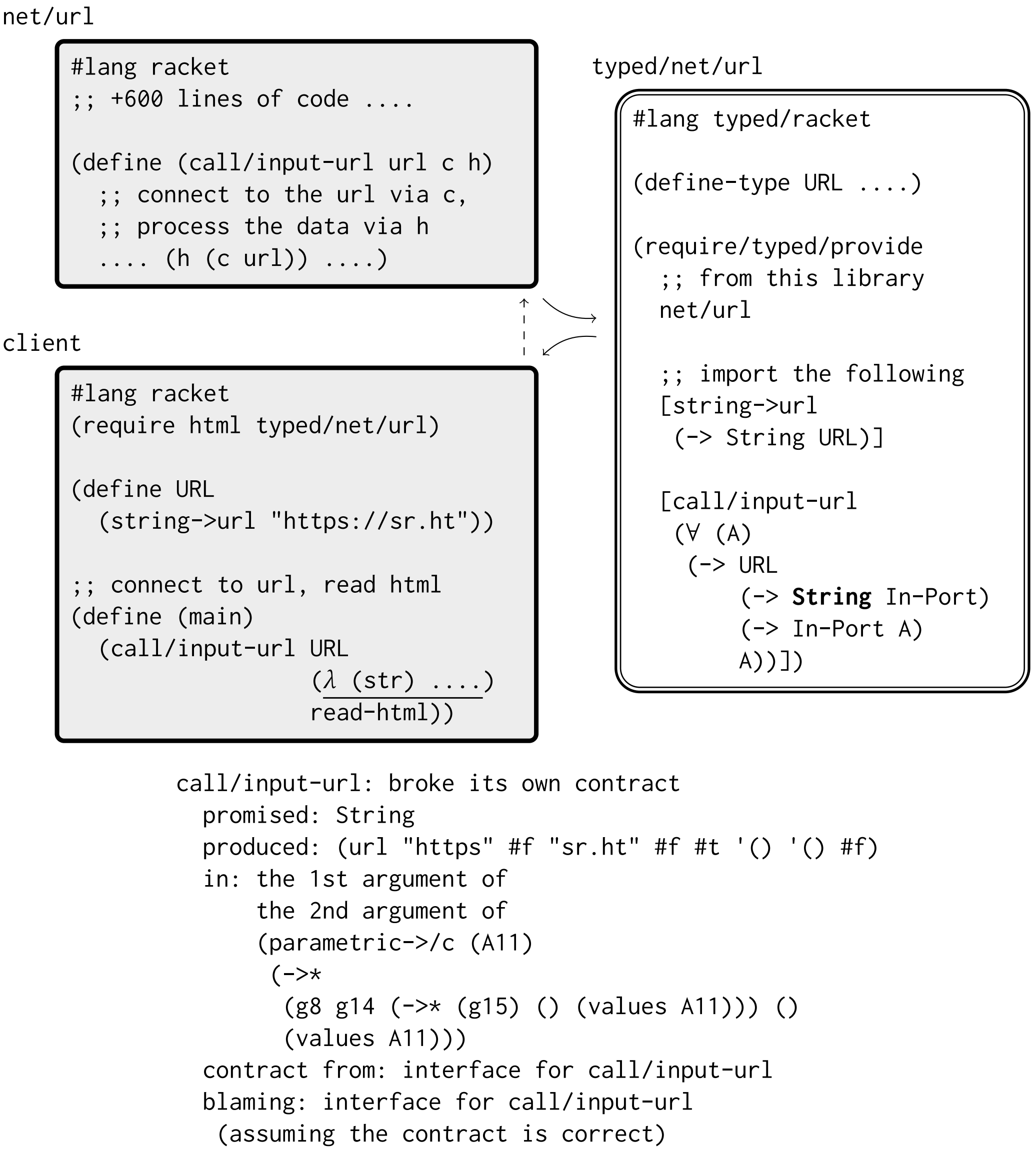
The second example is about pairs. It asks what happens when typed code declares a pair type and receives an untyped pair:

$$\boxed{g = \lambda(x : \mathsf{Int}{\times}\mathsf{Int}).\,(\mathsf{fst}\ x) + 1} \rightleftarrows \boxed{g\ \langle\text{“}A\text{”}, 2\rangle} \tag{2}$$

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 substantially 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) directly because the language does not allow truly untyped 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. Instead, programmers must declare one kind of pair for every two types they wish to combine. 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 programming burden. 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 of this rigidity, the model in section 6 supports neither Nom nor other concrete languages [19, 52, 61, 93],



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
  .... (h (c url)) ....)
```

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))
```

```
call/input-url: broke its own contract
  promised: String
  produced: (url "https" #f "sr.ht" #f #t '() '() #f)
  in: the 1st argument of
      the 2nd argument of
      (parametric->/c (A11)
       (->*
        (g8 g14 (->* (g15) () (values A11))) ()
        (values A11)))
  contract from: interface for call/input-url
  blaming: interface for call/input-url
   (assuming the contract is correct)
```

Fig. 4. Typed Racket detects and reports a higher-order type mismatch

Both Reticulated and Typed Racket raise an error on program (2), but for 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 and receives a string. In general, Typed Racket eagerly checks the contents of data structures while Reticulated lazily validates them at use-sites.

2.3 Debugging Higher-Order Interactions

Figures 4 and 5 present simplified excerpts from realistic programs that mix typed and untyped code. These examples follow a common general structure: an untyped client interacts with an untyped library through a thin layer of typed code. The solid arrows indicate these statically visible dependencies. Additionally, the untyped client supplies an argument to the untyped service module that, due to type annotations, dynamically opens a back channel to the client; the dashed arrow indicates this dynamic dependency of the two untyped modules. Both programs also happen to signal run-time errors, but do so for different reasons and with rather different implications.

The first example shows how Typed Racket's implementation of the Natural semantics, which monitors all interactions that cross type boundaries, can detect a mistake in a type declaration. The second example uses Reticulated's implementation of the Transient semantics to demonstrate how a type-sound language can fail to detect a mismatch between a value and a type.

2.3.1 A Mistaken Type Declaration. Figure 4 consists of an untyped library, an *incorrect* layer of type annotations, and an untyped client of the typed layer. The module at the top left, `net/url`, is a snippet from an untyped library that has been part of Racket for two decades.[5] The typed module on the right defines types for part of the library. Lastly, the module at the bottom left imports the typed library and invokes 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 layer. The client application clearly relies on the type specification from `typed/net/url` based on the arguments that it sends: the first is a URL structure, the second (underlined) 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 boldface type **`String`** in figure 4 is in conflict with the code in the library, which applies the second argument (a function) of `call/input-url` to a URL struct rather than a string.

Fortunately, 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 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." Thus, the contract error is a warning that either the code in `net/url` or the type in its interface is incorrect; and indeed, the type from which the contract is derived is an incorrect specification of the library's behavior.

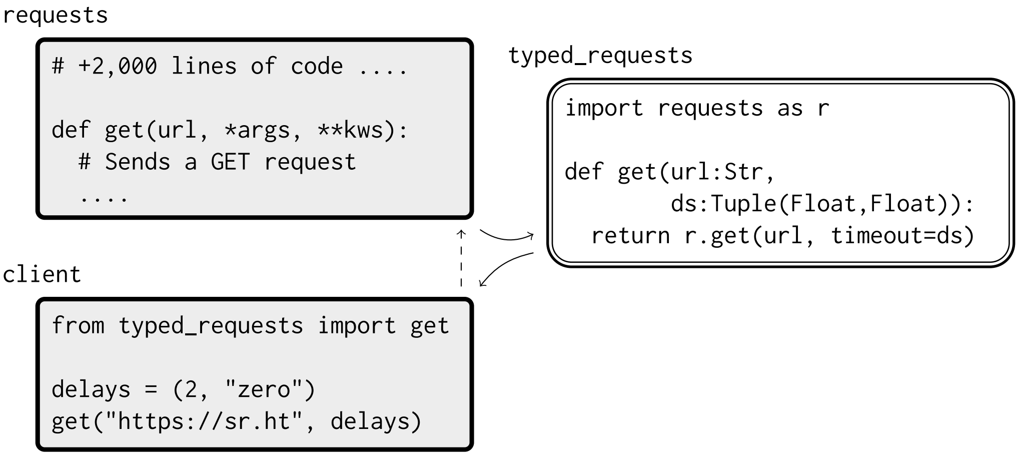
Alternative Possibility. If Typed Racket was merely type-sound, it would not be guaranteed to catch the type mismatch between the interface and the client. In this case, the client function (underlined) passed to `call/input-url` would be executed with a URL struct bound to the `str` variable. The consequences of this bad input would depend on how the function is implemented. If an error occurs at all, it might happen in the client and it might happen in another module that the function passes its input to. Either way, the typed module would be off the stack for the error message; programmers would have to remember its role to debug the type mistake.

2.3.2 A Data Structure Mismatch. 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.[6] This function accepts several optional and keyword arguments. The typed

---

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

[6] `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,
        ds:Tuple(Float,Float)):
  return r.get(url, timeout=ds)
```

client

```
from typed_requests import get

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

```
Traceback (most recent call last):
  File "client.py", line 81, in <module>
    get("https://sr.ht", (5, "zero"))
  File "typed_requests.py", line 23, in get
    return mgd_cast_type_function(
             cast0(requests, gensym2, '4', gensym3).get,
             gensym4, '4', gensym5)(url, timeout=timeout)
  File "/PythonLib/requests/api.py", line 75, in get
    return request('get', url, params=params, **kwargs)
  File "/PythonLib/requests/api.py", line 60, in request
    return session.request(method=method, url=url, **kwargs)
  File "/PythonLib/requests/sessions.py", line 533, in request
    resp = self.send(prep, **send_kwargs)
  File "/PythonLib/requests/sessions.py", line 646, in send
    r = adapter.send(request, **kwargs)
  File "/PythonLib/requests/adapters.py", line 436, in send
    raise ValueError(err)
ValueError: Invalid timeout (5, 'zero').
            Pass a (connect, read) timeout tuple, or a single float
            to set both timeouts to the same value
```

Fig. 5. Reticulated does not catch errors that occur in untyped Python code

adaptor module on the right formulates types for one valid use of the function; namely, 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 run-time checks ensure that the typed function receives a string and a tuple, but do not validate the tuple's contents. These same arguments thus pass to the untyped `get` function in the `requests` module. When the untyped `get` eventually uses the string `"zero"` as a float, Python (not Reticulated) 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.

In this example, the developer is lucky because the call to the typed version of `get` is still visible in 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 not be present.

Alternative Possibility. If Reticulated chose to traverse the bad tuple at the type boundary, it would discover the type mismatch. Similarly, if Reticulated checked all reads from the tuple in untyped contexts, it could detect the mismatch and raise an appropriate error. Both alternatives go beyond what is strictly required for type soundness, but would help for debugging this program.

3 COMPARING SEMANTICS

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 statically typed in a conventional sense or a weaker one; what typed APIs mean for untyped client code; and whether an error message can pinpoint which type specification does not match which value. All decisions have implications for language designers and programmers.

The examples in section 2 illustrate that various languages choose different points in this design space. But, examples can only motivate a systematic analysis; they cannot serve as an analysis. After all, examples tell us little about the broader implications of each choice.

A systematic analysis needs a suite of formal properties that differentiate the design choices for the language designer and working developer. These properties must apply to a large part of the design space. Finally, they should clarify which guarantees type specifications offer to the developers of typed and untyped code, respectively. While the literature focuses on type soundness and the blame theorem, our analysis adds new properties to the toolbox, which all parties should find helpful in making design choices or selecting languages for a project.

3.1 Type Soundness and the Blame Theorem

Type soundness is one formal property that meets the above criteria. A type soundness theorem can be tailored to a range of type systems, has meaning for typed and untyped code, and can be proven via a syntactic technique that scales to a variety of language features [92]. The use of type soundness in the literature, however, does not promote informed comparisons. Consider the four example languages from the previous section. Chaudhuri et al. [16] present a model of Flow and prove a conventional type soundness theorem under the assumption that all code is statically-typed. Vitousek et al. [86] prove a type soundness theorem for Reticulated Python that focuses on *shapes* of values rather than types. Muehlboeck and Tate [52] prove a full type soundness theorem for Nom. Tobin-Hochstadt and Felleisen [78] prove a full type soundness theorem for a prototypical Typed Racket that includes a weak blame property. These four 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.

Another well-studied property is the *blame theorem* [1, 64, 78, 86–88]. It states that a run-time mismatch may occur only when an untyped—or less-precisely typed—value enters a typed context. The property is a useful design principle, but too many languages satisfy this property too easily.

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, however, be clearly separated. For one, the canonical forms lemmas that support these different levels of soundness set limits on the type-directed optimizations that a compiler may safely perform.

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, whether every interaction between typed and untyped code is mediated by run-time checks. Section 2.3 illustrates this point with two contrasting example. Both open channels of communication between untyped pieces of code at run time—see the dashed arrows in figures 4 and 5—that are due to value flows through typed pieces of code. While Typed Racket's type-enforcement mechanism catches this problem, Reticulated's does not. (The problem is caught by the run-time checks of Python.)

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 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 of how many errors an enforcement regime discovers. The answer is given by 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. Additionally, this work should help with the investigation of the consequences of design choices for the working developer.

4 EVALUATION FRAMEWORK

To formulate different type-enforcement stategies on an equal footing, the framework is based on a single mixed-typed surface language (section 4.1). This syntax is then equipped with distinct semantics to model the different type-enforcement strategies (section 4.2). Type soundness (section 4.3) and complete monitoring (section 4.4) characterize the type mismatches that a semantics can detect. Blame soundness and blame completeness (section 4.5) measure the theoretical quality of error messages. The error preorder (section 4.6) is a direct comparison of the semantics.

4.1 Surface Language

The surface syntax is a multi-language that combines two independent pieces in the style of Matthews and Findler [48]. 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 uni-type. Boundary expressions connect the two pieces.

The uni-type $\mathcal{U}$ is not the flexible dynamic type from the theory of gradual typing that can replace any static type [5, 68, 77], rather, it describes all well-formed untyped expressions [48].[7] There is consequently no need for a type precision judgment in the surface language, because all typed–untyped interactions occur through boundary expressions. In this way, our surface language closely resembles the cast calculi that serve as intermediate languages in the gradual typing literature, e.g., [67, 69].

The sets of statically-typed (v_s) and dynamically-typed (v_d) values consist of integers, natural numbers, pairs, and functions:

$$v_s \;=\; i \mid n \mid \langle v_s, v_s\rangle \mid \lambda(x:\tau).\, e_s \qquad\qquad \tau \;=\; \mathsf{Int} \mid \mathsf{Nat} \mid \tau{\Rightarrow}\tau \mid \tau{\times}\tau$$

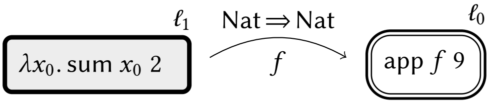
$$v_d \;=\; i \mid n \mid \langle v_d, v_d\rangle \mid \lambda x.\, e_d$$

---

[7] 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.

These core value sets are relatively small, but they suffice to illustrate the behavior of 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 adds some realism to the model[8] and allows it to distinguish between two sound enforcing methods (declaration-site vs. use-site).

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

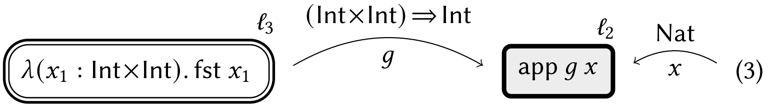


(3)

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 the server module ℓ_1, henceforth called *sender*:

$$(3)\ \simeq\ \mathsf{app}\ (\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \Rightarrow \mathsf{Nat} \blacktriangleleft \ell_1)\ (\lambda x_0.\,\mathsf{sum}\ x_0\ 2))\ 9$$

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 (3) on the right.



(4)

When linearized to the surface language, this term becomes:

$$\begin{aligned}(4)\ \simeq\ \mathsf{app}\ &(\mathsf{stat}\ (\ell_2 \blacktriangleleft \mathsf{Int}\times\mathsf{Int} \Rightarrow \mathsf{Int} \blacktriangleleft \ell_3)\ (\lambda(x_1 : \mathsf{Int}\times\mathsf{Int}).\,\mathsf{fst}\ x_1))\\ &(\mathsf{stat}\ (\ell_2 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_0)\ (3))\end{aligned}$$

Technically, a boundary expression combines a boundary specification b and a sender expression. A dyn boundary embeds dynamically-typed code in a typed context; a stat boundary embeds statically-typed code in an untyped context.[9] The specification includes the names of the interacting components 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 (i.e. $\ell_0 \in \ell$). The boundary types guide the static type checker, but are mere suggestions unless a semantics decides to enforce them:

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

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

---

[8] Adding this form of subtyping also ensures model can scale to include true union types, which are an integral part of the idiomatic type systems added to untyped languages [15, 80, 81].

[9] Boundary terms are similar to casts from the gradual typing literature, but provide more structure for blame assignment. A boundary connects a typed component to an untyped component. A cast connects typed code to less-precisely typed code; both sides of a cast may be part of the same component.

code. Conversely, a well-formed untyped expression may include any typed expression that matches the specified annotation:

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

Each surface-language component must have a name. These names must be *coherent* in the sense that the *client* name in all boundary specifications must match the name of its enclosing context.

The purpose of the names is to support blame assignment when an typed–untyped interaction goes wrong. Suppose a program halts due to a mismatch between a type Nat and a value −2. If the semantics has knowledge of both the client and sender of the bad value, then an error report can include this boundary where Nat is required and −2 arrived.

4.2 Semantic Framework

The first ingredient a reduction semantics must supply is the set of result values v to which expressions may reduce. Our result sets extend the sets of core values introduced in the preceding subsection ($v \supseteq v_s \cup v_d$). Potential reasons for extending the value set include the following:

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

Reasons 1 and 2 call for two kinds of wrapper value.[10] A guard wrapper ($\mathbb{G}\ b\ v$) associates a boundary specification with a value to achieve delayed type checks. Guards are similar to boundary expressions; they separate a context component from a value component. A trace wrapper ($\mathbb{T}\ \overline{b}\ v$) attaches a list of boundaries to a value as metadata. Trace wrappers simply annotate values.

The second ingredient is a set of notions of reduction, most importantly those for boundary expressions. For example, the Natural semantics (section 6.5) fully enforces types via the classic wrapper techniques [25, 48], which is expressed as follows where a filled triangle (►) describes a step in untyped code and an open triangle (▷) describes a step in typed code:

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

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

According to the first rule, a typed number may enter an untyped context without further ado. According to the second rule, typed code may access an untyped function only through a newly-created guard wrapper. Guard wrappers are a *higher-order* tool for enforcing types for first-class functions. As such, wrappers require elimination rules. To complete its type-enforcement strategy, the Natural semantics includes the following rule to unfold the application of a guarded function into two boundaries:

$$\begin{gathered}\mathsf{app}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ (\lambda x_0.\, -8))\ 1 \rhd_{\mathsf{N}} \\ \mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (\mathsf{app}\ (\lambda x_0.\, -8)\ (\mathsf{stat}\ (\ell_1 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_0)\ 1))\end{gathered} \tag{c}$$

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 of using wrappers, it

---

[10] A language with the dynamic type will need a third wrapper for base values that have been assigned type dynamic.

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

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

The triangle is filled gray ($\rhd$) because Transient is defined via a single 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 properties of the result. Since there are two kinds of surface expressions, soundness has two parts: one for statically-typed code and one for dynamically-typed code.

For typed code, the question is the extent to which surface types predict the result of an evaluation. There are a range of possible answers. Suppose that an expression with surface type τ_0 reduces to a value. 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 merely a well-formed value, with no stronger prediction about its shape. Even in this weak extreme, however, the language guarantees that typed reductions cannot reach an undefined state.

For untyped code, there is one surface type. Soundness guarantees that evaluation cannot reach an undefined state, but it cannot predict the shape of result values.

Both parts combine into the following definition, where the function F and judgment $\vdash_F$ are parameters. The function F 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. For example, for Natural, F is the identify function and for Transient, it is a function that ignores all but the top-level constructor of a type. The judgment $\vdash_F$ matches a value with a description.

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* $\vdash_F v_0 : F(\tau_0)$
- e_0 *reduces to an allowed error*
- e_0 *diverges.*

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* $\vdash_F v_0 : \mathcal{U}$
- e_0 *reduces to an allowed error*
- e_0 *diverges.*

4.4 Complete Monitoring

The complete monitoring property holds if a language has complete control over every *type-induced* channel of communication between two components in a world that mixes typed and untyped code. Consider an identity function that flows from an untyped component ℓ_0 to a typed one ℓ_1, through an ($\mathsf{Int} \Rightarrow \mathsf{Int}$) type annotation. Now imagine that this function flows into untyped component ℓ_2, which applies this function to itself. This application opens a channel of communication between ℓ_0 and ℓ_2 at run time. This channel is *type-induced* because the identity function migrated to this point through a type boundary. If the language satisfies complete monitoring, it rejects this application because the argument is a function and not an integer; an error report could point back to the boundary between ℓ_0 and ℓ_1, which imposed the obligation that arguments must be of type Int.

At first glance, this example seems to inject sophistication where none is needed. In particular, applying the identity function to itself does no harm. But, as section 2.3 explains with a distilled real-world example, such mis-applications can be the result of type specifications for untyped code that are simply wrong. Thus, while the type checker may bless the typed code, its interactions with untyped code may reveal the mismatch between the obligation that a type imposes and the computations that the code performs.

Our approach to validating complete monitoring uses the well-known subject-reduction technique for a semantics modified to track obligations imposed by type boundaries. Tracking these obligations relies on tracking boundary crossings via component labels, dubbed *ownership labels* by Dimoulas et al. [22]. A sequence of labels on a value reflects the path that the value has taken through components and, by implication, which type obligations the value has incurred. These labels enrich the semantics with information *without changing it*. A meta-type system describes desired properties of the evaluation in terms of the labels, and subject reduction establishes that the properties hold.

Labels track information as follows. At the start of an evaluation, no interactions have occurred yet and every expression has exactly one label that names the component in which it resides. When a boundary term reduces, an interaction happens and the labels in the result term change as follows:

- If the sender component supplies a value whose adherence to a client's type specification can be fully checked, then the value loses its old labels and comes under full control of the client.
- If the check has to be partial, because the value is higher-order, there are two possible outcomes depending on how the value crosses the boundary:
  - If the original value crosses over as is, it keeps its old labels and acquires the labels of the client. The sender and client share joint responsibility for the value going forward.
  - If the client receives a newly-created proxy, then the proxy acquires the client's labels and the wrapped value retains its old labels. The sender remains responsible for the wrapped value, and the client has full responsibility for the proxy.

  In short, the ownership labels on a value denotes the parties responsible for the behavior of the value. Storing these labels as a sequence keeps track of the order in which they gained responsibility for the value.

A semantics that prevents joint-responsibility situations satisfies the goal of complete monitoring; it controls every typed–untyped interaction. When a language is in control, it can present useful error messages as demonstrated in section 2.3.1. When a language is not in control, misleading errors can arise due to issues at type boundaries as the example in section 2.3.2 illustrates.

An ownership label $^{\ell_0}$ 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 $^{\ell_0}$ and $(((42)^{\ell_0})^{\ell_1})^{\ell_2}$—short-hand: $((42))^{\ell_0\ell_1\ell_2}$—is an integer with three owners.

A complete monitoring theorem requires two ingredients that manage these labels. First, a reduction relation $\rightarrow^{*}_{\mathrm{r}}$ 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 relative to a map $\mathcal{L}_0$ from variables to their binding component. To satisfy complete monitoring, reduction must preserve single-ownership.

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

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

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

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.

The definition of complete monitoring states that a labeled reduction relation must preserve the single-ownership invariant.

Definition Sketch (complete monitoring).
For all $\cdot;\ell_0 \Vdash e_0$*, any reduction* $e_0 \rightarrow_{\mathbf{r}}^{*} e_1$ *implies* $\cdot;\ell_0 \Vdash e_1$*.*

4.4.1 How to Uniformly Equip a Reduction Relation with Labels.

In practice, a language comes with an unlabeled reduction system, and it is up to a researcher to design a lifted relation that propagates labels without changing the underlying relations. Lifting thus requires insight. If labels do not transfer correctly, then a complete monitoring theorem loses (some of) its meaning. Similarly, if the behavior of a lifted relation depends on labels, then a theorem about it does not apply to the original, un-lifted reduction system.

Section 6 present six reduction relations as the semantics of our single mixed-typed syntax. Each relation needs a lifted version to support an attempt at a complete monitoring theorem. Normally, the design of any lifted reduction relation is a challenge in itself [22, 23, 51, 76]. Labels must reflect the communications that arise at run-time, and the possible communications depend on the unlabeled semantics. The six lifted relations for this paper, however, follow a common pattern. Section 6 therefore presents one lifted relation as an example (section 6.5) and defers to the supplementary material for the others.

To give readers an intuition for how each lifted relation comes about, this section presents informal guidelines for managing labels in a path-based way. Each guideline describes one way that labels may be transferred or dropped during evaluation and comes with an illustrative reduction.

Because labels are an analytical tool that (in principle) apply to any reduction relation, the examples are posed in terms of a *hypothetical* reduction relation $\mathbf{r}$ over the surface language. To read an example, assume the unlabeled notion of reduction $e\ \mathbf{r}\ e$ is given and focus on how the labels (superscripts) change in response. Recall that stat and dyn are boundary terms; they link two different components, a client context and an enclosed sender expression, via a type.

(G1) If a base value reaches a boundary with a matching base type, then the value drops its current labels as it crosses the boundary.
Example: $(\mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (0)^{\ell_2\ell_1})^{\ell_0}\ \mathbf{r}\ (0)^{\ell_0}$
Explanation: The value 0 fully matches the type Nat.

(G2) Otherwise, a value that crosses a boundary acquires the label of the new component.
Example: $(\mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (\langle -2, 1\rangle)^{\ell_1})^{\ell_0}\ \mathbf{r}\ ((\langle -2, 1\rangle))^{\ell_1\ell_0}$
Explanation: The pair $\langle -2, 1\rangle$ does not match the type Nat.

(G3) Every value that flows out of a value v_0 acquires the labels of v_0 and the context.
Example: $(\mathsf{snd}\ ((\langle (1)^{\ell_0}, (2)^{\ell_1}\rangle))^{\ell_2\ell_3})^{\ell_4}\ \mathbf{r}\ ((2))^{\ell_1\ell_2\ell_3\ell_4}$
Explanation: The value 2 flows out of the pair $\langle 1, 2\rangle$.

(G4) Every value that flows into a function v_0 acquires the context's label and v_0's reversed labels.
Example: $(\mathsf{app}\ ((\lambda x_0.\,\mathsf{fst}\ x_0))^{\ell_0\ell_1}\ (\langle 8, 6\rangle)^{\ell_2})^{\ell_3}\ \mathbf{r}\ (((\mathsf{fst}\ ((\langle 8, 6\rangle))^{\ell_2\ell_3\ell_1\ell_0}))^{\ell_0\ell_1})^{\ell_3}$
Explanation: The argument value $\langle 8, 6\rangle$ is input to the function. The substituted body flows out of the function, and by G3 acquires the function's labels.

(G5) A new value produced by a primitive acquires the context's label.
Example: $(\mathsf{sum}\ (2)^{\ell_0}\ (3)^{\ell_1})^{\ell_2}\ \mathbf{r}\ (5)^{\ell_2}$
Explanation: Ignoring the labels, $\delta(\mathsf{sum}, 2, 3) = 5$.

(G6) Consecutive equal labels are dropped; they do not represent boundary crossings.
Example: $((0))^{\ell_0\ell_0\ell_1\ell_0} = ((0))^{\ell_0\ell_1\ell_0}$

(G7) Labels on an error term are dropped; the path of an error term is not important.
Example: $(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_1)\ (\mathsf{sum}\ 9\ (\mathsf{DivErr})^{\ell_1}))^{\ell_0}\ \ \mathbf{r}\ \ \mathsf{DivErr}$

Although guideline G4 refers specifically to functions, the concept generalizes to reference cells and to other values that accept inputs.

To demonstrate how these guidelines influence a lifted reduction relation, the following rules lift the examples from section 4.2. Each rule accepts input with any sequence of labels ($\bar{\ell}$), pattern-matches the important labels, and shuffles labels in accordance with the guidelines. The first rule (a') demonstrates a base-type boundary (G1). The second (b') demonstrates a higher-order boundary (G2); the new guard on the right-hand side implicitly inherits the context label. The third rule (c') sends an input (G4) and creates new application and boundary expressions. The fourth rule (d') applies G3 for an output.

$$
\begin{array}{llr}
(\mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ ((42))^{\bar{\ell}_2})^{\ell_3} & \blacktriangleright_{\overline{\mathsf{N}}}\ \ (42)^{\ell_3} & (a') \\
(\mathsf{dyn}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ ((\lambda x_0.\ ((-8))^{\bar{\ell}_2}))^{\bar{\ell}_3})^{\ell_4} & \triangleright_{\overline{\mathsf{N}}} & (b') \\
\qquad (\mathbb{G}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ ((\lambda x_0.\ ((-8))^{\bar{\ell}_2}))^{\bar{\ell}_3})^{\ell_4} & & \\
(\mathsf{app}\ ((\mathbb{G}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ (v_0)^{\ell_2}))^{\bar{\ell}_3}\ ((1))^{\bar{\ell}_4})^{\ell_5} & \triangleright_{\overline{\mathsf{N}}} & (c') \\
\qquad (\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (\mathsf{app}\ (v_0)^{\ell_2}\ (\mathsf{stat}\ (\ell_1 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_0)\ ((1))^{\bar{\ell}_4\ell_5 rev(\bar{\ell}_3)}))^{\ell_2})^{\bar{\ell}_3\ell_5} & & \\
(\mathsf{check}\{(\mathsf{Nat} \times \mathsf{Nat})\}\ ((\langle((-1))^{\bar{\ell}_0}, ((-2))^{\bar{\ell}_1}\rangle))^{\bar{\ell}_2})^{\ell_3} & \triangleright_{\overline{\mathsf{T}}}\ \ ((\langle((-1))^{\bar{\ell}_0}, ((-2))^{\bar{\ell}_1}\rangle))^{\bar{\ell}_2\ell_3} & (d')
\end{array}
$$

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. To begin debugging, a programmer should know which boundaries the value traversed; after all, it is these boundaries that imposed the violated obligations. A semantics may offer information by blaming a set of boundaries. Then the 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. Ownership labels let us check whether the set b_0^* has anything to do with the boundaries that the lifted semantics recorded, that is, the sequence of labels attached to the v_0 value. Relative to this source-of-truth, blame soundness asks whether the names in b_0^* are a subset of the labels. Blame completeness asks for a superset of the labels.

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 technical challenge is to either satisfy both or find a middle ground.

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.*

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*.*

4.6 Error Preorder

Whereas the above properties characterize semantics independently of one another, the *error preorder relation* sets up a direct comparison. One semantics is below another in this preorder, written $X \lesssim Y$, if it raises errors on at least as many well-formed programs. Put another way, $X \lesssim Y$ holds when X is less permissive than Y is. When two semantics agree about which expressions raise run-time errors, we use the notation $X \eqsim Y$.

Definition Sketch (error preorder $\lesssim$).
$X \lesssim Y$ *iff* $e_0 \rightarrow_{\mathsf{Y}}^{*} \mathsf{Err}$ *implies* $e_0 \rightarrow_{\mathsf{X}}^{*} \mathsf{Err}$.

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

The six semantics in this paper are especially close to one another. Although they use different methods for enforcing types, they agree on other behaviors. In particular, these semantics diverge on the same expressions and compute equivalent values ignoring wrappers. This close correspondence lets us view the error preorder in another way: $X \lesssim Y$ holds for these semantics if and only if Y reduces at least as many expressions to a result value ($\{e_0 \mid \exists v_0.\, e_0 \rightarrow_{\mathsf{X}}^{*} v_0\} \subseteq \{e_1 \mid \exists v_1.\, e_1 \rightarrow_{\mathsf{Y}}^{*} v_1\}$). The supplementary material presents bisimulations that establish the correspondences.

5 TYPE-ENFORCEMENT STRATEGIES

The six chosen type-enforcement strategies share some commonalities and exhibit significant differences in philosophy and technicalities. This section supplies the ideas behind each strategy and serves as a quick, informal reference. Readers who prefer formal definitions may wish to skip to section 6.

The overview begins with the strategy that is lowest on the error preorder and ascends to the most lenient strategy:

Natural	Wrap higher-order values; eagerly check 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	Never use wrappers; check the shape of all values that appear in typed code.
Amnesic	Check shapes like Transient; use wrappers only to remember boundary types.
Erasure	Never use wrappers; check nothing. Do not enforce static types at run-time.

Three of these strategies have been implemented in full-fledged languages: Natural, Transient, and Erasure. Two, Co-Natural and Forgetful, originate in prior work [31, 34] and, sitting between the Natural and Transient strategies, highlight the variety of designs. Finally, Amnesic is a synthetic semantics, created to demonstrate how the analysis framework can be used to address problems, specifically the impoverished nature of blame assignment in Transient.

5.1 Natural

Natural strictly enforces the boundaries between typed and untyped code. Every time a typed context imports an untyped value, the value undergoes a comprehensive check. For first-order values, this implies a deep traversal of the incoming value. For higher-order values, a full check at the time of crossing the boundary means creating a wrapper to monitor its future behavior.

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

dyn Int v ▷ · check that v is an integer	stat Int v ▶ · check nothing
dyn $(\tau_0 \times \tau_1)$ v ▷ · check that v is a tuple and recursively validate its elements	stat $(\tau_0 \times \tau_1)$ v ▶ · recursively protect the elements
dyn $(\tau_0 \Rightarrow \tau_1)$ v ▷ · check that v is a function and wrap v to protect higher-order inputs and validate outputs	stat $(\tau_0 \Rightarrow \tau_1)$ v ▶ · wrap v to validate inputs and protect higher-order outputs

Fig. 6. Natural boundary checks (omitting blame)

Figure 6 describes in more detail the checks that happen when a value reaches a boundary. The descriptions omit component names and blame in order to keep the focus on types. These checks either validate an untyped value entering typed code (left column) or protect a typed value before it enters untyped code (right column).

5.1.1 Theoretical Costs, Motivation for Alternative Methods.

Implementations of Natural have struggled with the performance overhead of enforcing types [25, 38]. 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 and other first-order values, 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 whose cost is linear in the size of the type. To check recursive types such as 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 [24, 31, 41, 44, 74].

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.

Researchers have addressed these costs to some extent with implementation techniques that lower the time and space bounds for Natural [6, 14, 24, 31, 41, 44, 63, 66] without changing its behavior. The next three type-enforcement strategies can, however, offer more drastic improvements. First, the Co-Natural strategy (section 5.2) reduces the up-front cost of checks by creating wrappers for pairs. 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 type soundness invariant.

5.1.2 Origins of the Natural strategy.

The name "Natural" is due to Matthews and Findler [48], who use it to describe a proxy method for transporting untyped functions into a typed context. Prior works on higher-order contracts [25], remote procedure calls [55], and typed foreign function interfaces [56] employ a similar type-directed proxy method. In the gradual typing literature, Natural is also called "guarded" [84], "behavioral" [18], and "deep" [82]. This strategy has an

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

dyn Int $v \;\triangleright\; \cdot$ check that v is an integer	stat Int $v \;\blacktriangleright\; \cdot$ check nothing
dyn $(\tau_0 \times \tau_1)$ $v \;\triangleright\; \cdot$ check that v is a tuple and wrap v to validate its elements	stat $(\tau_0 \times \tau_1)$ $v \;\blacktriangleright\; \cdot$ wrap v to protect its elements
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 $(\tau_0 \Rightarrow \tau_1)$ $v \;\blacktriangleright\; \cdot$ wrap v to validate inputs and protect higher-order outputs

Fig. 7. Co-Natural boundary checks

interesting justification via work on AGT [28]; namely, its checks ensure that a proof of type preservation is still possible given the untyped values that have arisen at runtime.

5.2 Co-Natural

The Co-Natural strategy 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 even first-order values are wrapped in monitors. Figure 7 outlines the strategy.

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 the data structure is accessed to perform a check. The choice is analogous to the question of initial algebra vs. final algebra semantics for such datatypes [7, 12, 89], hence the prefix "Co" is a reminder that some checks now happen at an opposite time. Findler et al. [27] implement exactly the Co-Natural strategy for Racket struct contracts. Other researchers have explored variations on lazy contracts [17, 20, 21, 42]; for instance, by delaying even shape checks until a computation depends on the value.

5.3 Forgetful

The goal of Forgetful is to guarantee type soundness and to limit the number of wrappers around a value. A non-goal is to enforce types in any way that is not strictly required by soundness. Consequently, types in Forgetful are *not* compositionally-valid claims about code. Typed code can rely on the static types that it declares, nothing more. Untyped code cannot trust type annotations because those types may be forgotten without ever getting 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. The sticky wrapper protects the function from bad inputs. The temporary outer wrappers protect callers. Figure 8 presents an outline of the strategy.

5.3.1 Comparison to Natural. Figure 9 present two examples to demonstrate how Forgetful manages guard wrappers as compared to the Natural semantics.[11] Each example term sends an identity

---

[11] Since these examples use only function types, they exhibit the same behavior according to Co-Natural as well as Natural.

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

dyn Int v $\rhd$ $\cdot$ check that v is an integer	stat Int v $\blacktriangleright$ $\cdot$ check nothing
dyn $(\tau_0 \times \tau_1)$ v $\rhd$ $\cdot$ check that v is a tuple and wrap v to validate its elements	stat $(\tau_0 \times \tau_1)$ v $\blacktriangleright$ $\cdot$ if v has a wrapper, discard it; otherwise wrap v to protect its elements
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	stat $(\tau_0 \Rightarrow \tau_1)$ v $\blacktriangleright$ $\cdot$ if v has a wrapper, discard it; otherwise wrap v to validate inputs and protect higher-order outputs

Fig. 8. Forgetful boundary checks

function across three boundaries. To keep the illustration concise, let A, B, and C be three types such that the example terms are well-typed. The three boundaries at hand use the function types A ⇒ A, B ⇒ B, and C ⇒ C.

These examples are formatted in a tabular layout. Each row of the table corresponds to a type-enforcement strategy. From left to right, the cells in a row show how a value accumulates guard wrappers. Each column states whether the current redex is untyped or typed. Untyped columns have a shaded background. Typed columns come with an expected type. Similarly, the arrows between the columns are open (▷) when the value passes through a dyn boundary and filled (▶) when the value passes through a stat boundary. The top of each figure presents a full example term that can be reduced using the semantics in section 6.

Example: Untyped Identity Function. Figure 9 (top) shows how Natural and Forgetful add wrappers to an untyped function that crosses three boundaries. Natural creates one wrapper for each boundary. Forgetful creates a temporary wrapper whenever the function enters a typed context and removes this wrapper when the function exits.

Example: Typed Identity Function. Figure 9 (bottom) shows how Natural and Forgetful add wrappers to a typed function that crosses three boundaries. Natural creates one guard wrapper for each boundary. Forgetful creates an initial "sticky" guard wrapper when a typed function first exits typed code. This wrapper enforces the function's domain type. When the function re-enters typed code, Forgetful adds a wrapper to record its new type. When it exits typed code, this outer wrapper gets forgotten.

5.3.2 Origins of the Forgetful strategy. Greenberg [30, 31] introduces forgetful manifest contracts, proves their type soundness, and observes that unlike normal types, forgetful types cannot support abstraction and information hiding. Castagna and Lanvin [15] present a gradual language with union and intersection types that has a forgetful semantics to keep the formalism simple without affecting type soundness.

There are other strategies that limit the number of wrappers on a value without sacrificing type guarantees [31, 41, 63]. These methods require an implementation of wrappers that can be merged with one another, whereas Forgetful can treat wrappers as black boxes.

An untyped function crosses three boundaries:

$\mathsf{dyn}\ (C \Rightarrow C)\ (\mathsf{stat}\ (B \Rightarrow B)\ (\mathsf{dyn}\ (A \Rightarrow A)\ \lambda x_0.\, x_0))$

	$\mathcal{U}$	▷	$A \Rightarrow A$	▶	$\mathcal{U}$	▷	$C \Rightarrow C$
Natural	$\lambda x_0.\, x_0$		$\mathbb{G}\ (A \Rightarrow A)$ $(\lambda x_0.\, x_0)$		$\mathbb{G}\ (B \Rightarrow B)$ $\mathbb{G}\ (A \Rightarrow A)$ $(\lambda x_0.\, x_0)$		$\mathbb{G}\ (C \Rightarrow C)$ $\mathbb{G}\ (B \Rightarrow B)$ $\mathbb{G}\ (A \Rightarrow A)$ $(\lambda x_0.\, x_0)$
Forgetful	$\lambda x_0.\, x_0$		$\mathbb{G}\ (A \Rightarrow A)$ $(\lambda x_0.\, x_0)$		$\lambda x_0.\, x_0$		$\mathbb{G}\ (C \Rightarrow C)$ $\lambda x_0.\, x_0$

A typed function crosses three boundaries:

$\mathsf{stat}\ (C \Rightarrow C)\ (\mathsf{dyn}\ (B \Rightarrow B)\ (\mathsf{stat}\ (A \Rightarrow A)\ \lambda (x_0 : A).\, x_0))$

	$A \Rightarrow A$	▷	$\mathcal{U}$	▶	$B \Rightarrow B$	▷	$\mathcal{U}$
Natural	$\lambda (x_0 : A).\, x_0$		$\mathbb{G}\ (A \Rightarrow A)$ $\lambda (x_0 : A).\, x_0$		$\mathbb{G}\ (B \Rightarrow B)$ $\mathbb{G}\ (A \Rightarrow A)$ $\lambda (x_0 : A).\, x_0$		$\mathbb{G}\ (C \Rightarrow C)$ $\mathbb{G}\ (B \Rightarrow B)$ $\mathbb{G}\ (A \Rightarrow A)$ $\lambda (x_0 : A).\, x_0$
Forgetful	$\lambda (x_0 : A).\, x_0$		$\mathbb{G}\ (A \Rightarrow A)$ $\lambda (x_0 : A).\, x_0$		$\mathbb{G}\ (B \Rightarrow B)$ $\mathbb{G}\ (A \Rightarrow A)$ $\lambda (x_0 : A).\, x_0$		$\mathbb{G}\ (A \Rightarrow A)$ $\lambda (x_0 : A).\, x_0$

Fig. 9. Natural vs. Forgetful

5.4 Transient

The Transient strategy aims to prevent typed code from "going wrong" [49] in the sense of applying a primitive operation to a value outside its domain. For example, every application $(e_0\ e_1)$ in Transient-typed code can trust that the value of e_0 is a function.

Transient meets this goal without wrappers and without traversing data structures by rewriting typed code ahead-of-time in a conservative fashion. Every type boundary, every typed elimination form, and every typed function body gets rewritten to execute a shape check. These shape checks match the top-level constructor of a value against the top-level constructor of a type. By applying shape checks wherever an ill-typed value might sneak in, Transient protects typed code against undefined primitive operations.

Figure 10 describes the checks that happen at a boundary in the Transient semantics. Unlike the other semantics, however, these boundary checks are only part of the story. Additional dyn-style checks appear within typed code because of the rewriting pass.

In general, Transient checks add up to a greater number of run-time validation points than those that arise in a wrapper-based semantics because every expression in typed code may require a check. The net cost of these checks, however, may be lower and easier to predict than in higher-order strategies because each check has a low cost [29, 37, 62, 86]. Often a tag check suffices, although unions and other expressive types require a deeper check [36]. Static analysis can further reduce

Transient strategy. 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 pair
dyn $(\tau_0 \Rightarrow \tau_1)\ v \ \triangleright\ \cdot$
check that v is a function

stat Int $v \ \blacktriangleright\ \cdot$
check nothing
stat $(\tau_0 \times \tau_1)\ v \ \blacktriangleright\ \cdot$
check nothing
stat $(\tau_0 \Rightarrow \tau_1)\ v \ \blacktriangleright\ \cdot$
check nothing

Typed code dyn-checks outputs of elimination forms and inputs to functions.

Fig. 10. Transient boundary checks

costs by identifying overly-conservative checks [85], and JIT compilers have been effective at reducing the costs of Transient [29, 46, 62, 85]

5.4.1 Origins of the Transient strategy. Vitousek [83] invented Transient for Reticulated Python. The name suggests the nature of its run-time checks: Transient type-enforcement enforces local assumptions in typed code but has no long-lasting ability to influence untyped behaviors [84]. Transient has been adapted to Typed Racket [34, 36] and has inspired closely-related approaches in Grace [29, 62] and in Static Python [46].

5.5 Amnesic

The goal of the Amnesic semantics is to specify basically 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 wraps values, discards all but three wrappers, and keeps a record of discarded boundary specifications. To record boundaries, Amnesic uses *trace* wrappers. When a type mismatch occurs, Amnesic presents the recorded boundaries to the programmer.

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. Figure 11 outlines the strategy.

5.5.1 Comparison to Forgetful and Transient. The design of Amnesic is best understood as a variation of Transient that accepts a limited number of wrappers per value. Like the Forgetful semantics, it puts at most two guard wrappers around a value. It also uses at most one trace wrapper to remember all boundaries that the value has crossed.

The following two examples compare Forgetful, Transient, and Amnesic side-by-side using the same example terms as in figure 9. As before, let A ⇒ A, B ⇒ B, and C ⇒ C be three function types such that the example terms are well-typed.

Example: Untyped Identity Function. Figure 12 (top) shows how Forgetful, Transient, and Amnesic manage an untyped function that crosses three boundaries. Forgetful creates a wrapper when the function enters typed code and removes a wrapper when it leaves. Transient lets the function cross boundaries without creating wrappers. Amnesic creates the same guard wrappers as Forgetful and also uses a trace wrapper to record the obligations from forgotten guards.

Amnesic strategy. 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 its elements

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

Fig. 11. Amnesic boundary checks

Example: Typed Identity Function. Figure 12 (bottom) shows how Forgetful, Transient, and Amnesic manage a typed function that crosses three boundaries. Both Forgetful and Amnesic create a sticky wrapper when the function leaves typed code. When the function re-enters typed code, they add a second guard wrapper that gets removed on the next exit. Amnesic additionally uses a trace wrapper to collect all boundaries that the function has crossed. Transient does not create wrappers.

5.5.2 *Theoretical Costs.*

Amnesic is a theoretical design that may not be realizable in practice. In particular, an implementation must find an efficient representation of trace wrappers. Trace wrappers track every boundary that a value has crossed. Consequently, they have a space-efficiency problem similar to the unbounded number of guard wrappers in the Natural and Co-Natural semantics. One simple fix is to settle for worse blame by putting an upper bound on the number of boundaries that a trace wrapper can hold. Another option is to invent a compression scheme that exploits redundancies among boundaries to reduce the space needs of a large set.

5.5.3 *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 [35]. The name suggests a connection to forgetful and the Greek origin of the second author.

5.6 Erasure

The Erasure strategy 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 run-time, types check nothing (figure 13). Any value may flow into any context.

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 in 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 [50] and Common Lisp [71]. StrongTalk is another early and influential optionally-typed language [11]. Models of optional typing exist for JavaScript [8, 16], Lua [47], and Clojure [10].

An untyped function crosses three boundaries:

$$\mathsf{dyn}\ (C \Rightarrow C)\ (\mathsf{stat}\ (B \Rightarrow B)\ (\mathsf{dyn}\ (A \Rightarrow A)\ \lambda x_0.\, x_0))$$

	$\mathcal{U}$	▶	$A \Rightarrow A$	▷	$\mathcal{U}$	▶	$C \Rightarrow C$
Forgetful	$\lambda x_0.\, x_0$		$\mathbb{G}\,(A \Rightarrow A)$ $(\lambda x_0.\, x_0)$		$\lambda x_0.\, x_0$		$\mathbb{G}\,(C \Rightarrow C)$ $(\lambda x_0.\, x_0)$
Transient	$\lambda x_0.\, x_0$		$\lambda x_0.\, x_0$		$\lambda x_0.\, x_0$		$\lambda x_0.\, x_0$
Amnesic	$\lambda x_0.\, x_0$		$\mathbb{G}\,(A \Rightarrow A)$ $(\lambda x_0.\, x_0)$		$\mathbb{T}\,\{(B \Rightarrow B), (A \Rightarrow A)\}$ $\lambda x_0.\, x_0$		$\mathbb{G}\,(C \Rightarrow C)$ $\mathbb{T}\,\{(B \Rightarrow B), (A \Rightarrow A)\}$ $\lambda x_0.\, x_0$

A typed function crosses three boundaries:

$$\mathsf{stat}\ (C \Rightarrow C)\ (\mathsf{dyn}\ (B \Rightarrow B)\ (\mathsf{stat}\ (A \Rightarrow A)\ \lambda(x_0 : A).\, x_0))$$

	$A \Rightarrow A$	▷	$\mathcal{U}$	▶	$B \Rightarrow B$	▷	$\mathcal{U}$
Forgetful	$\lambda(x_0 : A).\, x_0$		$\mathbb{G}\,(A \Rightarrow A)$ $\lambda(x_0 : A).\, x_0$		$\mathbb{G}\,(B \Rightarrow B)$ $\mathbb{G}\,(A \Rightarrow A)$ $\lambda(x_0 : A).\, x_0$		$\mathbb{G}\,(A \Rightarrow A)$ $\lambda(x_0 : A).\, x_0$
Transient	$\lambda(x_0 : A).\, x_0$		$\lambda(x_0 : A).\, x_0$		$\lambda(x_0 : A).\, x_0$		$\lambda(x_0 : A).\, x_0$
Amnesic	$\lambda(x_0 : A).\, x_0$		$\mathbb{G}\,(A \Rightarrow A)$ $\lambda(x_0 : A).\, x_0$		$\mathbb{G}\,(B \Rightarrow B)$ $\mathbb{G}\,(A \Rightarrow A)$ $\lambda(x_0 : A).\, x_0$		$\mathbb{T}\,\{(C \Rightarrow C), (B \Rightarrow B)\}$ $\mathbb{G}\,(A \Rightarrow A)$ $\lambda(x_0 : A).\, x_0$

Fig. 12. Forgetful vs. Transient vs. Amnesic

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

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

Fig. 13. Erasure boundary checks

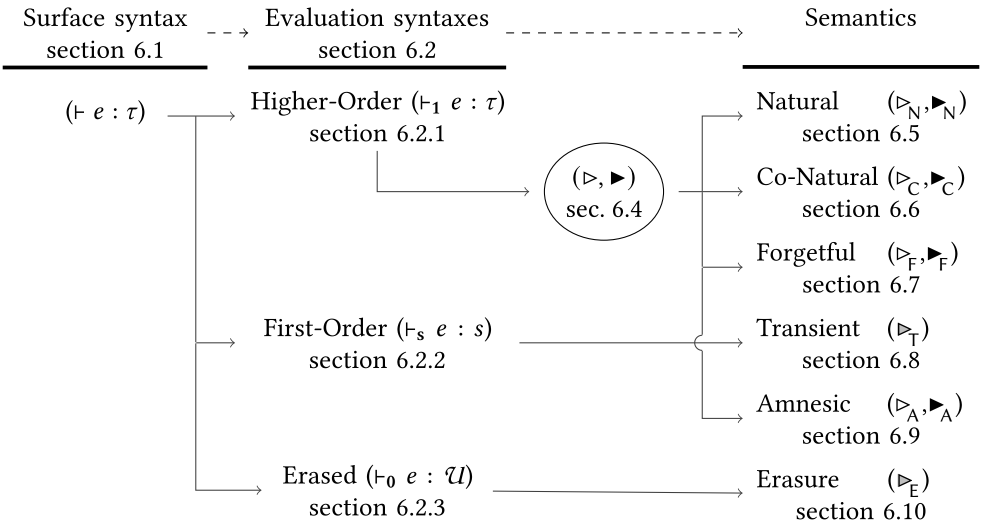


Fig. 14. Map of definitions in section 6

6 TECHNICAL DEVELOPMENT

The technical analysis consists of three major pieces: the precise surface syntax (section 6.1); the six reduction semantics, each equipped with a typed evaluation syntax (section 6.2); and a set of theorems concerning the properties that each semantics satisfies. Figure 14 displays a diagram that outlines the presentation. As the diagram indicates, four of the semantics share a common evaluation syntax; the intrinsically first-order transient semantics is separate from those.

Several properties depend on lifted semantics that propagate ownership labels in accordance with the guidelines from section 4.4.1. Meaning, the map in figure 14 is only half of the formal development. Each syntax and semantics comes with a parallel, lifted version. Since the differences are in small details, the section presents only one lifting in full. The others appear in the supplement.

6.1 Surface Syntax, Types, and Ownership

Figure 15 presents the syntax and typing judgments for the surface language. Expressions e include variables, integers, pairs, functions, primitive operations, applications, and boundary expressions. The primitive operations are pair projections and arithmetic functions; these model interactions with a runtime system. A boundary expression either embeds a dynamically-typed expression in a statically-typed context (dyn) or a typed expression in an untyped context (stat).

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 grammatical; the model describes two parallel syntaxes that are connected through boundary expressions (section 4.1). A statically-typed expression e_0 is one for which 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 $\mathsf{Nat} \leqslant: \mathsf{Int}$, covariant for pairs and function codomains, and 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 Γ_1.

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

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

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

$$\frac{(x_0 : \tau_0) \in \Gamma_0}{\Gamma_0 \vdash x_0 : \tau_0} \qquad \frac{}{\Gamma_0 \vdash n_0 : \mathsf{Nat}} \qquad \frac{}{\Gamma_0 \vdash i_0 : \mathsf{Int}} \qquad \frac{(x_0 : \tau_0), \Gamma_0 \vdash e_0 : \tau_1}{\Gamma_0 \vdash \lambda(x_0 : \tau_0).\, e_0 : \tau_0 \Rightarrow \tau_1}$$

$$\frac{\Gamma_0 \vdash e_0 : \tau_0 \qquad \Gamma_0 \vdash e_1 : \tau_1}{\Gamma_0 \vdash \langle e_0, e_1\rangle : \tau_0 \times \tau_1} \qquad \frac{\Gamma_0 \vdash e_0 : \tau_1 \qquad \Delta(\mathit{unop}, \tau_1) \leqslant: \tau_0}{\Gamma_0 \vdash \mathit{unop}\{\tau_0\}\; e_0 : \tau_0} \qquad \frac{\begin{array}{c}\Gamma_0 \vdash e_0 : \tau_1 \qquad \Gamma_0 \vdash e_1 : \tau_2 \\ \Delta(\mathit{binop}, \tau_1, \tau_2) \leqslant: \tau_0\end{array}}{\Gamma_0 \vdash \mathit{binop}\{\tau_0\}\; e_0\; e_1 : \tau_0}$$

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

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

$$\frac{(x_0 : \mathcal{U}) \in \Gamma_0}{\Gamma_0 \vdash x_0 : \mathcal{U}} \qquad \frac{}{\Gamma_0 \vdash i_0 : \mathcal{U}} \qquad \frac{(x_0 : \mathcal{U}), \Gamma_0 \vdash e_0 : \mathcal{U}}{\Gamma_0 \vdash \lambda x_0.\, e_0 : \mathcal{U}} \qquad \frac{\Gamma_0 \vdash e_0 : \mathcal{U} \qquad \Gamma_0 \vdash e_1 : \mathcal{U}}{\Gamma_0 \vdash \langle e_0, e_1\rangle : \mathcal{U}}$$

$$\frac{\Gamma_0 \vdash e_0 : \mathcal{U}}{\Gamma_0 \vdash \mathit{unop}\{\mathcal{U}\}\; e_0 : \mathcal{U}} \qquad \frac{\Gamma_0 \vdash e_0 : \mathcal{U} \qquad \Gamma_0 \vdash e_1 : \mathcal{U}}{\Gamma_0 \vdash \mathit{binop}\{\mathcal{U}\}\; e_0\; e_1 : \mathcal{U}} \qquad \frac{\Gamma_0 \vdash e_0 : \mathcal{U} \qquad \Gamma_0 \vdash e_1 : \mathcal{U}}{\Gamma_0 \vdash \mathsf{app}\{\mathcal{U}\}\; e_0\; e_1 : \mathcal{U}}$$

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

Fig. 15. Surface syntax and typing rules

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 reconstruct valid annotations from the term and its context. 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 16 augments 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 augmented 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

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

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

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

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

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

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

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

$$\frac{\mathcal{L}_0; \ell_0 \Vdash e_0 \qquad \mathcal{L}_0; \ell_0 \Vdash e_1}{\mathcal{L}_0; \ell_0 \Vdash \mathsf{app}\{{}^{\tau}\!/_{\mathcal{U}}\}\ e_0\ e_1} \qquad \frac{\mathcal{L}_0; \ell_1 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash \mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ (e_0)^{\ell_1}} \qquad \frac{\mathcal{L}_0; \ell_1 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash \mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ (e_0)^{\ell_1}}$$

$$\frac{\mathcal{L}_0; \ell_0 \Vdash e_0}{\mathcal{L}_0; \ell_0 \Vdash (e_0)^{\ell_0}}$$

Fig. 16. Ownership syntax and single-owner consistency

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 map entry for that variable; 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 18), is inconsistent for any parameters.

Labels correspond one-to-one 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 (${}^{\ell_1}$). The label matches the inner component name, which means that the inner component is responsible for the variable inside the boundary. The reason for using two distinct sets is to keep our analysis framework separate from the semantics that it analyzes. Whereas a semantics can freely inspect and manipulate component names (which would be realized as symbols or addresses in an implementation), it cannot use labels to determine its behavior (labels would not be part of an implementation).

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

6.2 Three Evaluation Syntaxes

Each semantics requires a unique evaluation syntax, but overlaps among these six languages motivate three common platforms. A *higher-order* evaluation syntax supports type-enforcement strategies that require wrappers. A *first-order* 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.

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

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

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

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

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

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

Fig. 17. Common evaluation syntax and metafunctions

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

$$\mathit{rev}\,(\ell_0 \cdots \ell_n) = \ell_n \cdots \ell_0$$

$$\mathit{senders}\,(b_0^{*}) = \{\ell_1 \mid (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1) \in b_0^{*}\}$$

$$\mathit{owners}\,(v_0) = \begin{cases} \{\ell_0\} \cup \mathit{owners}\,(v_1) & \text{if } v_0 = (v_1)^{\ell_0} \\ \mathit{owners}\,(v_1) & \text{if } v_0 = \mathbb{T}\;b_0^{*}\;v_1 \\ \{\} & \text{otherwise} \end{cases}$$

$$\text{Abbreviation:}\quad ((e_0))^{\ell_n \cdots \ell_1} = e_1 \quad\Longleftrightarrow\quad e_1 = (\cdots (e_0)^{\ell_n} \cdots)^{\ell_1}$$

Fig. 18. Metafunctions for boundaries and labels

Figure 17 defines common aspects of the evaluation syntax. These include errors Err, shapes (or, constructors) s, evaluation contexts, and evaluation metafunctions.

The evaluation syntax *extends* the surface syntax in a technical sense; namely, the grammar presented in figure 17 would be complete if it included a copy of the grammar from figure 15. Every occurrence of the word "extends" in a figure has a similar meaning. For example, the typing judgments in figure 19 would be complete if the judgment rules from figure 15 were copied in.

A program evaluation may signal four kinds of errors.

- A dynamic tag error (TagErr) occurs when an elimination form is applied to a mis-shaped input. For example, the first projection of an integer signals a tag error.
- An invariant error (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. A type-sound system eliminates such contradictions.
- A division-by-zero error (DivErr) may be raised by an application of the quotient primitive. In a full language, there will be many additional primitive errors.
- A boundary error (BoundaryErr (b^*, v)) reports a mismatch between two components. The sender provides the enclosed value; the client rejects it. The set of witness boundaries suggests potential sources for the fault; intuitively, this set should include the client–sender boundary. The error 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.

  Remark: The semantics in this paper all blame a set of boundaries in order to share a common evaluation syntax. Many semantics can, however, provide more precise blame. Natural and Co-Natural can blame a single boundary; Forgetful and Amnesic can blame a sequence. The supplementary material presents these alternatives. In the supplement, it is therefore crucial that a lifted reduction relation tracks sequences of labels rather than sets.

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 numbers. By convention, a variable without a subscript refers to a set and a term containing a set describes a comprehension. The term $(\lambda(x : \tau).\, v)$, for instance, describes the set $\{(\lambda(x_i : \tau_j).\, v_k) \mid x_i \in x \wedge \tau_j \in \tau \wedge v_k \in v\}$ of all typed functions that return a value (rather than an expression).

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. Figure 19 formally introduces these wrapper values.

The final components of figure 17 are the δ metafunctions. These provide a standard and partial specification of the primitive operations.

Figure 18 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 $(\!(\cdot)\!)^{\cdot}$ 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 19) 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.[12] A trace wrapper $(\mathbb{T}\; b^*\; v)$ attaches metadata to a value.

---

[12] Correction note: our prior work uses the name *monitor wrapper* and value constructor mon [34, 35]. The name *guard wrapper* better matches earlier work [23, 76], in which `mon` creates an expression and `G` creates a wrapper.

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

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

$\boxed{\Gamma \vdash_1 e : \tau}$ extends $\Gamma \vdash e : \tau$ to allow guard wrappers and errors

$$\frac{\Gamma_0 \vdash_1 v_0 : \mathcal{U}}{\Gamma_0 \vdash_1 \mathbb{G}\,(\ell_0\blacktriangleleft\tau_0\blacktriangleleft\ell_1)\; v_0 : \tau_0} \qquad \frac{}{\Gamma_0 \vdash_1 \mathsf{Err} : \tau_0}$$

$\boxed{\Gamma \vdash_1 e : \mathcal{U}}$ extends $\Gamma \vdash e : \mathcal{U}$ to allow guard wrappers, trace wrappers, and errors

$$\frac{\Gamma_0 \vdash_1 v_0 : \tau_0}{\Gamma_0 \vdash_1 \mathbb{G}\,(\ell_0\blacktriangleleft\tau_0\blacktriangleleft\ell_1)\; v_0 : \mathcal{U}} \qquad \frac{\Gamma_0 \vdash_1 v_0 : \mathcal{U}}{\Gamma_0 \vdash_1 \mathbb{T}\, b_0^*\, v_0 : \mathcal{U}} \qquad \frac{}{\Gamma_0 \vdash_1 \mathsf{Err} : \mathcal{U}}$$

$\boxed{\mathcal{L};\ell \Vdash e}$ extends $\mathcal{L};\ell \Vdash e$ to allow guard wrappers and trace wrappers

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

Fig. 19. Higher-Order syntax, typing rules, and ownership consistency

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, $\Gamma \vdash_1 e : {}^{\tau}\!/_{\mathcal{U}}$, 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 28). A traced expression is well-formed iff the enclosed value is well-formed. An error term is well-typed in any context.

Figure 19 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 First-order Syntax. The first-order syntax (figure 20) supports typed–untyped interaction without proxy wrappers. A new expression form, $(\mathsf{check}\{{}^{\tau}\!/_{\mathcal{U}}\}\; e_0\; \mathsf{p}_0)$, 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_0 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 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 first-order typing judgments state basic invariants. For statically-typed expressions, the judgment checks the top-level shape (s) of an expression and the well-formedness of any subexpressions. This judgment depends on a subtyping judgment for shapes, which is reflexive, allows $\mathsf{Nat} \leqslant: \mathsf{Int}$,

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

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

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

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

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

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

$\boxed{\mathcal{T};\Gamma \vdash_{\mathsf{s}} e : s}$

$$\frac{(\mathsf{p}_0 : s_0) \in \mathcal{T}_0}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} \mathsf{p}_0 : s_0} \qquad \frac{(x_0 : \tau_0) \in \Gamma_0}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} x_0 : \lfloor\tau_0\rfloor} \qquad \frac{}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} i_0 : \mathsf{Int}} \qquad \frac{}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} n_0 : \mathsf{Nat}}$$

$$\frac{\mathcal{T}_0;(x_0 : \mathcal{U}),\Gamma_0 \vdash_{\mathsf{s}} e_0 : \mathcal{U}}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} \lambda x_0.\, e_0 : \mathsf{Fun}} \qquad \frac{\mathcal{T}_0;(x_0 : \tau_0),\Gamma_0 \vdash_{\mathsf{s}} e_0 : s_0}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} \lambda(x_0 : \tau_0).\, e_0 : \mathsf{Fun}} \qquad \frac{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} e_0 : s_0 \qquad \mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} e_1 : s_1}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} \langle e_0, e_1\rangle : \mathsf{Pair}}$$

$$\frac{}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} \mathsf{Err} : s_0} \qquad \frac{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} e_0 : \mathsf{Fun} \qquad \mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} e_1 : s_0}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} \mathsf{app}\{\tau_0\}\, e_0\, e_1 : \lfloor\tau_0\rfloor} \qquad \frac{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} e_0 : \mathsf{Pair}}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} \mathit{unop}\{\tau_0\}\, e_0 : \lfloor\tau_0\rfloor}$$

$$\frac{\begin{array}{c}\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} e_0 : s_0 \qquad \mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} e_0 : s_1 \\ \Delta(\mathit{binop}, s_0, s_1) = \tau_1 \qquad \tau_1 \leqslant: \tau_0\end{array}}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} \mathit{binop}\{\tau_0\}\, e_0\, e_1 : \lfloor\tau_0\rfloor} \qquad \frac{\mathcal{T}_0;\Gamma_0 \vdash e_0 : \mathcal{U}}{\mathcal{T}_0;\Gamma_0 \vdash \mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ e_0 : \lfloor\tau_0\rfloor}$$

$$\frac{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} e_0 : \mathcal{U}}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} \mathsf{check}\{\tau_0\}\, e_0\, \mathsf{p}_0 : \lfloor\tau_0\rfloor} \qquad \frac{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} e_0 : s_0}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} \mathsf{check}\{\tau_0\}\, e_0\, \mathsf{p}_0 : \lfloor\tau_0\rfloor} \qquad \frac{\begin{array}{c}\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} e_0 : s_1 \\ s_1 \leqslant: s_0\end{array}}{\mathcal{T}_0;\Gamma_0 \vdash_{\mathsf{s}} e_0 : s_0}$$

$\boxed{\mathcal{T};\Gamma \vdash_{\mathsf{s}} e : \mathcal{U}}$ selected rules that handle references, variables, boundaries, and checks

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

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

Fig. 20. First-order syntax and typing rules

and nothing more. For dynamically-typed expressions, the judgment checks well-formedness. Both judgments rely on a store typing environment ($\mathcal{T}$) to describe heap-allocated values. Store types must be consistent with the actual values on the heap, a standard technical device that is spelled out in the supplement.

$\boxed{\text{Erased Evaluation Syntax}}$ 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 that handle variables, functions, and boundaries

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

$$\frac{\Gamma_0 \vdash_{\mathbf{0}} e_0 : \mathcal{U} \qquad \Gamma_0 \vdash_{\mathbf{0}} e_1 : \mathcal{U}}{\Gamma_0 \vdash_{\mathbf{0}} \mathsf{app}\{\mathcal{U}\}\, e_0\, e_1 : \mathcal{U}} \qquad \frac{\Gamma_0 \vdash_{\mathbf{0}} e_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathbf{0}} \mathsf{dyn}\, b_0\, e_0 : \mathcal{U}} \qquad \frac{\Gamma_0 \vdash_{\mathbf{0}} e_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathbf{0}} \mathsf{stat}\, b_0\, e_0 : \mathcal{U}}$$

Fig. 21. Erased evaluation syntax and typing

Two aspects of the first-order typing judgments deserve special mention. First, untyped functions may appear in typed contexts and typed functions may appear in untyped contexts. This behavior is an essential aspect of the first-order language, which allows typed-untyped interoperability and does not use wrappers to enforce a separation between the two worlds. Second, shape-check expressions are allowed in typed and untyped contexts. This is a technical device. In particular, checks arise after a function call to separate the substituted body from the calling context, and this separation allows the typing judgments to switch from static mode to dynamic mode as needed.

6.2.3 Erased Syntax. Figure 21 defines an evaluation syntax for type-erased programs. Expressions include error terms. The typing judgment holds for any expression without free variables. Aside from 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.

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 an evaluation going wrong [49].

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

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

Three surface-to-evaluation maps (F) suffice for the evaluation languages: an identity map $\mathbf{1}$, a type-shape map $\mathbf{s}$ that extends the metafunction from figure 17, and a constant map $\mathbf{0}$:

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

Complete monitoring guarantees that a semantics can enforce types for all interactions between components. The definition of “all interactions” comes from the propagation guidelines (section 4.4.1). In particular, the labels on a value enumerate all partially-responsible components. Relative to this specification, a reduction that preserves single-owner consistency ($\Vdash$, figure 16) ensures that a value cannot enter a new component without a full type check or a wrapper.

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

$$
\begin{array}{lll}
\multicolumn{3}{l}{\boxed{e \rhd e}} \\
\mathit{unop}\{\tau_0\}\, v_0 & \rhd & \mathsf{InvariantErr} \\
\quad \text{if } v_0 \notin (\mathbb{G}\ (\ell \blacktriangleleft (\tau{\times}\tau) \blacktriangleleft \ell)\ v) & & \\
\quad \text{and } \delta(\mathit{unop}, v_0) \text{ is undefined} & & \\
\mathit{unop}\{\tau_0\}\, v_0 & \rhd & \delta(\mathit{unop}, v_0) \\
\quad \text{if } \delta(\mathit{unop}, v_0) \text{ is defined} & & \\
\mathit{binop}\{\tau_0\}\, v_0\, v_1 & \rhd & \mathsf{InvariantErr} \\
\quad \text{if } \delta(\mathit{binop}, v_0, v_1) \text{ is undefined} & & \\
\mathit{binop}\{\tau_0\}\, v_0\, v_1 & \rhd & \delta(\mathit{binop}, v_0, v_1) \\
\quad \text{if } \delta(\mathit{binop}, v_0, v_1) \text{ is defined} & & \\
\mathsf{app}\{\tau_0\}\, v_0\, v_1 & \rhd & \mathsf{InvariantErr} \\
\quad \text{if } v_0 \notin (\lambda(x : \tau).\, e) \cup & & \\
\qquad\qquad (\mathbb{G}\ (\ell \blacktriangleleft (\tau {\Rightarrow} \tau) \blacktriangleleft \ell)\ v) & & \\
\mathsf{app}\{\tau_0\}\, v_0\, v_1 & \rhd & e_0[x_0 \leftarrow v_1] \\
\quad \text{if } v_0 = (\lambda(x_0 : \tau_1).\, e_0) & &
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\boxed{e \blacktriangleright e}} \\
\mathit{unop}\{\mathcal{U}\}\, v_0 & \blacktriangleright & \mathsf{TagErr} \\
\quad \text{if } v_0 \notin (\mathbb{G}\ (\ell \blacktriangleleft (\tau{\times}\tau) \blacktriangleleft \ell)\ v) & & \\
\quad \text{and } \delta(\mathit{unop}, v_0) \text{ is undefined} & & \\
\mathit{unop}\{\mathcal{U}\}\, v_0 & \blacktriangleright & \delta(\mathit{unop}, v_0) \\
\quad \text{if } \delta(\mathit{unop}, v_0) \text{ is defined} & & \\
\mathit{binop}\{\mathcal{U}\}\, v_0\, v_1 & \blacktriangleright & \mathsf{TagErr} \\
\quad \text{if } \delta(\mathit{binop}, v_0, v_1) \text{ is undefined} & & \\
\mathit{binop}\{\mathcal{U}\}\, v_0\, v_1 & \blacktriangleright & \delta(\mathit{binop}, v_0, v_1) \\
\quad \text{if } \delta(\mathit{binop}, v_0, v_1) \text{ is defined} & & \\
\mathsf{app}\{\mathcal{U}\}\, v_0\, v_1 & \blacktriangleright & \mathsf{TagErr} \\
\quad \text{if } v_0 \notin (\lambda x.\, e) \cup & & \\
\qquad\qquad (\mathbb{G}\ (\ell \blacktriangleleft (\tau {\Rightarrow} \tau) \blacktriangleleft \ell)\ v) & & \\
\mathsf{app}\{\mathcal{U}\}\, v_0\, v_1 & \blacktriangleright & e_0[x_0 \leftarrow v_1] \\
\quad \text{if } v_0 = (\lambda x_0.\, e_0) & &
\end{array}
$$

Fig. 22. Common notions of reduction for Natural, Co-Natural, Forgetful, and Amnesic

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 reports a subset of the true senders, though it may miss some or even all. A blame-complete semantics reports all the true senders, though it may also report irrelevant extras. A sound and complete semantics reports exactly the responsible components.

The path-based definitions for blame soundness and blame completeness rely on the propagation guidelines from section 4.4.1. Relative to these guidelines, the definitions relate the sender names in a set of boundaries (figure 18) 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 $e_0 \rightarrow^*_{\mathsf{X}} \mathsf{BoundaryErr}\,(b^*_0, v_0)$:*

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

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 X semantics is less permissive than the Y semantics (or, as section 4.6 notes, Y reduces at least as many expressions to a value as X).

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

If two semantics lie below one another according to 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 22. The rules for typed code ($\rhd$) handle elimination forms for unwrapped values and raise an invariant error ($\mathsf{InvariantErr}$) for invalid input. Type soundness ensures that such errors do not occur. The rules for

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

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

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

$$\begin{array}{lll}
\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\; v_0 & \rhd_{\mathsf{N}} & \mathbb{G}\,(\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\; v_0 \\
\quad \text{if } \textit{shape-match}\,(\lfloor \tau_0 \Rightarrow \tau_1 \rfloor, v_0) & & \\
\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\; \langle v_0, v_1\rangle & \rhd_{\mathsf{N}} & \langle \mathsf{dyn}\; b_0\; v_0, \mathsf{dyn}\; b_1\; v_1\rangle \\
\quad \text{if } \textit{shape-match}\,(\lfloor \tau_0 \times \tau_1 \rfloor, \langle v_0, v_1\rangle) & & \\
\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1) \text{ and } b_1 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1) & & \\
\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; i_0 & \rhd_{\mathsf{N}} & i_0 \\
\quad \text{if } \textit{shape-match}\,(\lfloor \tau_0 \rfloor, i_0) & & \\
\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 & \rhd_{\mathsf{N}} & \mathsf{BoundaryErr}\,(\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}, v_0) \\
\quad \text{if } \neg\textit{shape-match}\,(\lfloor \tau_0 \rfloor, v_0) & & \\
\mathsf{app}\{\tau_0\}\;(\mathbb{G}\,(\ell_0 \blacktriangleleft \tau_1 \Rightarrow \tau_2 \blacktriangleleft \ell_1)\; v_0)\; v_1 & \rhd_{\mathsf{N}} & \mathsf{dyn}\; b_0\; (\mathsf{app}\{\mathcal{U}\}\; v_0\; (\mathsf{stat}\; b_1\; v_1)) \\
\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1) \text{ and } b_1 = (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0) & &
\end{array}$$

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

$$\begin{array}{lll}
\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\; v_0 & \blacktriangleright_{\mathsf{N}} & \mathbb{G}\,(\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\; v_0 \\
\quad \text{if } \textit{shape-match}\,(\lfloor \tau_0 \rfloor, v_0) & & \\
\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\; \langle v_0, v_1\rangle & \blacktriangleright_{\mathsf{N}} & \langle \mathsf{stat}\; b_0\; v_0, \mathsf{stat}\; b_1\; v_1\rangle \\
\quad \text{if } \textit{shape-match}\,(\lfloor \tau_0 \times \tau_1 \rfloor, \langle v_0, v_1\rangle) & & \\
\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1) \text{ and } b_1 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1) & & \\
\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; i_0 & \blacktriangleright_{\mathsf{N}} & i_0 \\
\quad \text{if } \textit{shape-match}\,(\lfloor \tau_0 \rfloor, i_0) & & \\
\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 & \blacktriangleright_{\mathsf{N}} & \mathsf{InvariantErr} \\
\quad \text{if } \neg\textit{shape-match}\,(\lfloor \tau_0 \rfloor, v_0) & & \\
\mathsf{app}\{\mathcal{U}\}\;(\mathbb{G}\,(\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\; v_0)\; v_1 & \blacktriangleright_{\mathsf{N}} & \mathsf{stat}\; b_0\; (\mathsf{app}\{\tau_1\}\; v_0\; (\mathsf{dyn}\; b_1\; v_1)) \\
\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1) \text{ and } b_1 = (\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_0) & &
\end{array}$$

$\boxed{e \rightarrow^{*}_{\mathsf{N}} e}$ is the transitive, reflexive, compatible (with respect to evaluation contexts E, figure 17) closure of the relation $\bigcup\{\rhd_{\mathsf{N}}, \blacktriangleright_{\mathsf{N}}, \blacktriangleright, \rhd\}$

Fig. 23. Natural notions of reduction

untyped code (▶) raise a tag error for a malformed redex. Later definitions, for example figure 23, combine these relations (▷, ▶) with others to define a semantics.

6.5 Natural and its Properties

Figure 23 presents the values and key reduction rules for the Natural semantics. Conventional reductions handle primitives and unwrapped functions (▶ and ▷, figure 22).

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:

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

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* **TS (1)**.

PROOF SKETCH. By progress and preservation lemmas for the higher-order typing judgment ($\vdash_1$). For example, if an untyped pair reaches a boundary then a typed step ($\triangleright_{\mathsf{N}}$) makes progress to either a new pair or to an error. In the former case, the new pair contains two boundary expressions:

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

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

A second interesting case is for the rule that applies a wrapped function in a typed context:

$$\begin{array}{l}\mathsf{app}\{\tau_0\}\;(\mathbb{G}\;(\ell_0 \blacktriangleleft (\tau_1 \Rightarrow \tau_2) \blacktriangleleft \ell_1)\;v_0)\;v_1 \;\triangleright_{\mathsf{N}} \\ \quad \mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1)\;(\mathsf{app}\{\mathcal{U}\}\;v_0\;(\mathsf{stat}\;(\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_2)\;v_1))\end{array}$$

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.

□

Figure 24 presents a labeled variant of the Natural semantics for typed code. Ignoring labels, the rules in this figure are a combination of those in figures 22 and 23. The labels reflect communications and changes of ownership. The labeled rules for untyped code are similar and appear in the supplementary material.

THEOREM 6.7. *Natural satisfies* **CM**.

PROOF SKETCH. 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, which wraps a function. The lifted version ($\blacktriangleright_{\overline{\mathsf{N}}}$) accepts a term with arbitrary ownership labels and propagates these labels to the result:

$$\begin{array}{l}(\mathsf{stat}\;(\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\;((v_0))^{\overline{\ell_2}})^{\ell_3} \;\blacktriangleright_{\overline{\mathsf{N}}}\; (\mathbb{G}\;(\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\;((v_0))^{\overline{\ell_2}})^{\ell_3} \\ \quad \text{if } \textit{shape-match}\,(\lfloor \tau_0 \Rightarrow \tau_1 \rfloor, v_0)\end{array}$$

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 ($\overline{\ell_2} = \ell_1 \cdots \ell_1$). Under these premises, the result also satisfies single-owner consistency.

As a second example, consider the lifted rule that applies a wrapped function:

$$\begin{array}{l}(\mathsf{app}\{\tau_0\}\;((\mathbb{G}\;(\ell_0 \blacktriangleleft (\tau_1 \Rightarrow \tau_2) \blacktriangleleft \ell_1)\;(v_0)^{\ell_2}))^{\overline{\ell_3}}\;v_1)^{\ell_4} \;\triangleright_{\overline{\mathsf{N}}} \\ \quad (\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1)\;(\mathsf{app}\{\mathcal{U}\}\;v_0\;(\mathsf{stat}\;(\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0)\;(v_1)^{\ell_4 \mathit{rev}(\overline{\ell_3})}))^{\ell_2})^{\overline{\ell_3}\ell_4}\end{array}$$

$$\boxed{(e)^{\ell} \rhd_{\overline{\mathsf{N}}} (e)^{\ell}}\ \text{lifted version of } \rhd_{\mathsf{N}}$$

$$\begin{array}{ll}
(\mathit{unop}\{\tau_0\}\ ((v_0))^{\bar{\ell}_0})^{\ell_0} & \rhd_{\overline{\mathsf{N}}}\ (\mathsf{InvariantErr})^{\ell_0} \\
\quad \text{if } v_0 \notin (v)^{\ell} \text{ and } \delta(\mathit{unop}, v_0) \text{ is undefined} & \\
(\mathit{unop}\{\tau_0\}\ ((v_0))^{\bar{\ell}_0})^{\ell_0} & \rhd_{\overline{\mathsf{N}}}\ (\delta(\mathit{unop}, v_0))^{\bar{\ell}_0 \ell_0} \\
\quad \text{if } \delta(\mathit{unop}, v_0) \text{ is defined} & \\
(\mathit{binop}\{\tau_0\}\ ((v_0))^{\bar{\ell}_0}\ ((v_1))^{\bar{\ell}_1})^{\ell_0} & \rhd_{\overline{\mathsf{N}}}\ (\mathsf{InvariantErr})^{\ell_0} \\
\quad \text{if } v_0 \notin (v)^{\ell} \text{ and } v_1 \notin (v)^{\ell} \text{ and } \delta(\mathit{binop}, v_0, v_1) \text{ is undefined} & \\
(\mathit{binop}\{\tau_0\}\ ((v_0))^{\bar{\ell}_0}\ ((v_1))^{\bar{\ell}_1})^{\ell_0} & \rhd_{\overline{\mathsf{N}}}\ (\delta(\mathit{binop}, v_0, v_1))^{\ell_0} \\
\quad \text{if } \delta(\mathit{binop}, v_0, v_1) \text{ is defined} & \\
(\mathsf{app}\{\tau_0\}\ ((v_0))^{\bar{\ell}_0}\ v_1)^{\ell_0} & \rhd_{\overline{\mathsf{N}}}\ (\mathsf{InvariantErr})^{\ell_0} \\
\quad \text{if } v_0 \notin (v)^{\ell} \cup (\lambda x.\, e) \cup (\mathbb{G}\ b\ v) & \\
(\mathsf{app}\{\tau_0\}\ ((\lambda(x_0 : \tau_1).\, e_0))^{\bar{\ell}_0}\ v_1)^{\ell_0} & \rhd_{\overline{\mathsf{N}}}\ ((e_0[x_0 \leftarrow ((v_1))^{\ell_0\, \mathit{rev}(\bar{\ell}_0)}]))^{\bar{\ell}_0 \ell_0} \\
(\mathsf{app}\{\tau_0\}\ ((\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_1 \Rightarrow \tau_2 \blacktriangleleft \ell_1)\ (v_0)^{\ell_2}))^{\bar{\ell}_0}\ v_1)^{\ell_3} & \rhd_{\overline{\mathsf{N}}} \\
\qquad ((\mathsf{dyn}\ b_0\ (\mathsf{app}\{\mathcal{U}\}\ v_0\ (\mathsf{stat}\ b_1\ ((v_1))^{\ell_3\, \mathit{rev}(\bar{\ell}_0)}))^{\ell_2}))^{\bar{\ell}_0 \ell_3} & \\
\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1) \text{ and } b_1 = (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0) & \\
(\mathsf{dyn}\ (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\ ((v_0))^{\bar{\ell}_0})^{\ell_2} & \rhd_{\overline{\mathsf{N}}}\ (\mathbb{G}\ (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\ ((v_0))^{\bar{\ell}_0})^{\ell_2} \\
\quad \text{if } \mathit{shape\text{-}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))^{\bar{\ell}_0})^{\ell_2} & \rhd_{\overline{\mathsf{N}}}\ (\langle \mathsf{dyn}\ b_0\ ((v_0))^{\bar{\ell}_0}, \mathsf{dyn}\ b_1\ ((v_1))^{\bar{\ell}_0} \rangle)^{\ell_2} \\
\quad \text{if } \mathit{shape\text{-}match}(\lfloor \tau_0 \times \tau_1 \rfloor, \langle v_0, v_1 \rangle) \text{ and } b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1) \text{ and } b_1 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1) & \\
(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ ((i_0))^{\bar{\ell}_0})^{\ell_2} & \rhd_{\overline{\mathsf{N}}}\ (i_0)^{\ell_2} \\
\quad \text{if } \mathit{shape\text{-}match}(\lfloor \tau_0 \rfloor, i_0) & \\
(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ ((v_0))^{\bar{\ell}_0})^{\ell_2} & \rhd_{\overline{\mathsf{N}}}\ (\mathsf{BoundaryErr}\,((\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1), ((v_0))^{\bar{\ell}_0}))^{\ell_2} \\
\quad \text{if } \neg \mathit{shape\text{-}match}(\lfloor \tau_0 \rfloor, v_0) &
\end{array}$$

Fig. 24. Natural labeled notion of reduction for typed code

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

□

Blame soundness and completeness ask whether Natural identifies the components responsible for a boundary error. Here, complete monitoring helps to simplify the questions. Specifically, 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. Hence, every mismatch is due to a single boundary.

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

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

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

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

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

$$\begin{array}{lll}
\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 & \rhd_{\mathsf{C}} & \mathbb{G}\,(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \\
\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor\tau_0\rfloor, v_0) \text{ and } v_0 \in \langle v, v\rangle \cup (\lambda x.\, e) \cup (\mathbb{G}\, b\; v) & & \\
\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; i_0 & \rhd_{\mathsf{C}} & i_0 \\
\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor\tau_0\rfloor, i_0) & & \\
\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 & \rhd_{\mathsf{C}} & \mathsf{BoundaryErr}\,(\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}, v_0) \\
\quad \text{if } \neg\mathit{shape\text{-}match}\,(\lfloor\tau_0\rfloor, v_0) & & \\
\mathsf{fst}\{\tau_0\}\;(\mathbb{G}\,(\ell_0 \blacktriangleleft \tau_1 \times \tau_2 \blacktriangleleft \ell_1)\; v_0) & \rhd_{\mathsf{C}} & \mathsf{dyn}\; b_0\; (\mathsf{fst}\{\mathcal{U}\}\; v_0) \\
\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1) & & \\
\mathsf{snd}\{\tau_0\}\;(\mathbb{G}\,(\ell_0 \blacktriangleleft \tau_1 \times \tau_2 \blacktriangleleft \ell_1)\; v_0) & \rhd_{\mathsf{C}} & \mathsf{dyn}\; b_0\; (\mathsf{snd}\{\mathcal{U}\}\; v_0) \\
\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1) & & \\
\mathsf{app}\{\tau_0\}\;(\mathbb{G}\,(\ell_0 \blacktriangleleft \tau_1 \Rightarrow \tau_2 \blacktriangleleft \ell_1)\; v_0)\; v_1 & \rhd_{\mathsf{C}} & \mathsf{dyn}\; b_0\; (\mathsf{app}\{\mathcal{U}\}\; v_0\; (\mathsf{stat}\; b_1\; v_1)) \\
\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1) \text{ and } b_1 = (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0) & &
\end{array}$$

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

$$\begin{array}{lll}
\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 & \blacktriangleright_{\mathsf{C}} & \mathbb{G}\,(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \\
\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor\tau_0\rfloor, v_0) \text{ and } v_0 \in \langle v, v\rangle \cup (\lambda(x:\tau).\, e) \cup (\mathbb{G}\, b\; v) & & \\
\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; i_0 & \blacktriangleright_{\mathsf{C}} & i_0 \\
\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor\tau_0\rfloor, i_0) & & \\
\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 & \blacktriangleright_{\mathsf{C}} & \mathsf{InvariantErr} \\
\quad \text{if } \neg\mathit{shape\text{-}match}\,(\lfloor\tau_0\rfloor, v_0) & & \\
\mathsf{fst}\{\mathcal{U}\}\;(\mathbb{G}\,(\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\; v_0) & \blacktriangleright_{\mathsf{C}} & \mathsf{stat}\; b_0\; (\mathsf{fst}\{\tau_0\}\; v_0) \\
\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1) & & \\
\mathsf{snd}\{\mathcal{U}\}\;(\mathbb{G}\,(\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\; v_0) & \blacktriangleright_{\mathsf{C}} & \mathsf{stat}\; b_0\; (\mathsf{snd}\{\tau_1\}\; v_0) \\
\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1) & & \\
\mathsf{app}\{\mathcal{U}\}\;(\mathbb{G}\,(\ell_0 \blacktriangleleft \tau_0 \Rightarrow \tau_1 \blacktriangleleft \ell_1)\; v_0)\; v_1 & \blacktriangleright_{\mathsf{C}} & \mathsf{stat}\; b_0\; (\mathsf{app}\{\tau_1\}\; v_0\; (\mathsf{dyn}\; b_1\; v_1)) \\
\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1) \text{ and } b_1 = (\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_0) & &
\end{array}$$

$\boxed{e \rightarrow^{*}_{\mathsf{C}} e}$ is the transitive, reflexive, compatible (with respect to evaluation contexts E, figure 17) closure of the relation $\bigcup\{\rhd_{\mathsf{C}}, \blacktriangleright_{\mathsf{C}}, \blacktriangleright, \rhd\}$

Fig. 25. Co-Natural notions of reduction

$$\begin{array}{lll}
(e_0)^{\ell_0} & \rightarrow^{*}_{\mathsf{N}} & E[\mathsf{dyn}\;(\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\; v_0] \\
 & \rightarrow^{*}_{\mathsf{N}} & \mathsf{BoundaryErr}\,(\{(\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\}, v_0)
\end{array}$$

Thus $b^{*}_{0} = \{(\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\}$ and $\mathit{senders}\,(b^{*}_{0}) = \{\ell_2\}$. This boundary is the correct one to blame only if it matches the true owner of the value; that is, $\mathit{owners}\,(v_0) = \{\ell_2\}$. Complete monitoring guarantees a match via $\ell_0 \Vdash E[\mathsf{dyn}\;(\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\;(v_0)^{\ell_2}]$. ☐

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

6.6 Co-Natural and its Properties

Figure 25 presents the Co-Natural strategy. Co-Natural is a lazier 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 (1)**.

Proof Sketch. By progress and preservation lemmas for the higher-order typing judgment ($\vdash_1$). Many of the proof cases are similar to cases for Natural. One case unique to Co-Natural is for pairs that cross a boundary:

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

The typing rule for guard wrappers validates the result. ☐

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

Proof Sketch. By preservation of single-owner consistency for the lifted $\rightarrow^*_{\mathsf{C}}$ relation. For example, consider the lifted rule that extracts the first element from a wrapped, untyped pair:

$$(\mathsf{fst}\{\mathcal{U}\}\ ((\mathbb{G}\ (\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\ (v_0)^{\ell_2}))^{\overline{\ell}_3})^{\ell_4} \ \blacktriangleright_{\overline{\mathsf{C}}}\ (\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ (\mathsf{fst}\{\tau_0\}\ (v_0)^{\ell_2})^{\ell_2})^{\overline{\ell}_3 \ell_4}$$

If the redex satisfies single-owner consistency, then $\ell_0 = \overline{\ell}_3 = \ell_4$ and $\ell_1 = \ell_2$. ☐

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

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

Theorem 6.13. $N \lesssim C$.

Proof Sketch. 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 whereas Co-Natural creates a guard wrapper. Co-Natural takes additional steps when eliminating a wrapped pair. The supplement defines the simulation relation. ☐

Theorem 6.14. $C \not\lesssim N$.

Proof Sketch. The pair wrappers in Co-Natural imply $C \not\lesssim N$. 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 raise an error only if the first element is accessed. ☐

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

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

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

$$\begin{array}{ll}
\mathsf{dyn}\;(\ell_0\blacktriangleleft\tau_0\blacktriangleleft\ell_1)\; v_0 & \rhd_{\mathsf{F}}\;\; \mathbb{G}\;(\ell_0\blacktriangleleft\tau_0\blacktriangleleft\ell_1)\; v_0 \\
\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor\tau_0\rfloor, v_0) \text{ and } v_0 \in \langle v, v\rangle \cup (\lambda x.\, e) \cup (\mathbb{G}\, b\; v) & \\
\mathsf{dyn}\;(\ell_0\blacktriangleleft\tau_0\blacktriangleleft\ell_1)\; i_0 & \rhd_{\mathsf{F}}\;\; i_0 \\
\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor\tau_0\rfloor, i_0) & \\
\mathsf{dyn}\;(\ell_0\blacktriangleleft\tau_0\blacktriangleleft\ell_1)\; v_0 & \rhd_{\mathsf{F}}\;\; \mathsf{BoundaryErr}\,(\{(\ell_0\blacktriangleleft\tau_0\blacktriangleleft\ell_1)\}, v_0) \\
\quad \text{if } \neg\mathit{shape\text{-}match}\,(\lfloor\tau_0\rfloor, v_0) & \\
\mathsf{fst}\{\tau_0\}\;(\mathbb{G}\;(\ell_0\blacktriangleleft\tau_1\times\tau_2\blacktriangleleft\ell_1)\; v_0) & \rhd_{\mathsf{F}}\;\; \mathsf{dyn}\; b_0\;(\mathsf{fst}\{\mathcal{U}\}\; v_0) \\
\quad \text{where } b_0 = (\ell_0\blacktriangleleft\tau_1\blacktriangleleft\ell_1) & \\
\mathsf{snd}\{\tau_0\}\;(\mathbb{G}\;(\ell_0\blacktriangleleft\tau_1\times\tau_2\blacktriangleleft\ell_1)\; v_0) & \rhd_{\mathsf{F}}\;\; \mathsf{dyn}\; b_0\;(\mathsf{snd}\{\mathcal{U}\}\; v_0) \\
\quad \text{where } b_0 = (\ell_0\blacktriangleleft\tau_2\blacktriangleleft\ell_1) & \\
\mathsf{app}\{\tau_0\}\;(\mathbb{G}\;(\ell_0\blacktriangleleft\tau_1\Rightarrow\tau_2\blacktriangleleft\ell_1)\; v_0)\; v_1 & \rhd_{\mathsf{F}}\;\; \mathsf{dyn}\; b_0\;(\mathsf{app}\{\mathcal{U}\}\; v_0\;(\mathsf{stat}\; b_1\; v_1)) \\
\quad \text{where } b_0 = (\ell_0\blacktriangleleft\tau_2\blacktriangleleft\ell_1) \text{ and } b_1 = (\ell_1\blacktriangleleft\tau_1\blacktriangleleft\ell_0) &
\end{array}$$

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

$$\begin{array}{ll}
\mathsf{stat}\;(\ell_0\blacktriangleleft\tau_0\blacktriangleleft\ell_1)\; v_0 & \blacktriangleright_{\mathsf{F}}\;\; \mathbb{G}\;(\ell_0\blacktriangleleft\tau_0\blacktriangleleft\ell_1)\; v_0 \\
\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor\tau_0\rfloor, v_0) \text{ 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) & \blacktriangleright_{\mathsf{F}}\;\; v_0 \\
\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor\tau_0\rfloor, v_0) & \\
\quad \text{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 & \blacktriangleright_{\mathsf{F}}\;\; i_0 \\
\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor\tau_0\rfloor, i_0) & \\
\mathsf{stat}\;(\ell_0\blacktriangleleft\tau_0\blacktriangleleft\ell_1)\; v_0 & \blacktriangleright_{\mathsf{F}}\;\; \mathsf{InvariantErr} \\
\quad \text{if } \neg\mathit{shape\text{-}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) & \blacktriangleright_{\mathsf{F}}\;\; \mathsf{stat}\; b_0\;(\mathsf{fst}\{\tau_0\}\; v_0) \\
\quad \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) & \blacktriangleright_{\mathsf{F}}\;\; \mathsf{stat}\; b_0\;(\mathsf{snd}\{\tau_1\}\; v_0) \\
\quad \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 & \blacktriangleright_{\mathsf{F}}\;\; \mathsf{stat}\; b_0\;(\mathsf{app}\{\tau_1\}\; v_0\;(\mathsf{dyn}\; b_1\; v_1)) \\
\quad \text{where } b_0 = (\ell_0\blacktriangleleft\tau_1\blacktriangleleft\ell_1) \text{ and } b_1 = (\ell_1\blacktriangleleft\tau_0\blacktriangleleft\ell_0) &
\end{array}$$

$\boxed{e \rightarrow^{*}_{\mathsf{F}} e}$ is the transitive, reflexive, compatible (with respect to evaluation contexts E, figure 17) closure of the relation $\bigcup\{\rhd_{\mathsf{F}}, \blacktriangleright_{\mathsf{F}}, \blacktriangleright, \rhd\}$

Fig. 26. Forgetful notions of reduction

6.7 Forgetful and its Properties

The Forgetful semantics (figure 26) 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 protect 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. 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; in other words, Forgetful fails to satisfy blame completeness. Forgetful lies above Co-Natural and Natural in the error preorder because it fails to enforce certain type obligations.

Theorem 6.15. *Forgetful satisfies* **TS (1)**.

Proof Sketch. By progress and preservation lemmas for the higher-order typing judgment ($\vdash_1$). The most interesting proof case shows that dropping a guard wrapper does not break type preservation. 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:

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

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 dynamically-typed wrapper value with sufficient type information:

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

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

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

The new wrapper preserves types. □

Theorem 6.16. *Forgetful does not satisfy* **CM**.

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

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

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

Theorem 6.17. *Forgetful satisfies* **BS**.

Proof. By a preservation lemma for a weakened version of the $\Vdash$ judgment. The weak 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 unchecked label. Thus, when a boundary error occurs:

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

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

Theorem 6.18. *Forgetful does not satisfy* **BC**.

Proof. The proof of theorem 6.16 shows how a value can acquire two labels. If such a value triggers a boundary error, the error will be incomplete:

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

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

Theorem 6.19. $C \lesssim F$.

Proof Sketch. 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. □

Theorem 6.20. $F \not\lesssim C$.

Proof Sketch. $F \not\lesssim C$ because Forgetful drops checks. Let:

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

Then $e_1 \rightarrow^*_{\mathsf{F}} \langle 2, 8 \rangle$ and Co-Natural raises a boundary error. □

6.8 Transient and its Properties

The Transient semantics in figure 27 builds on the first-order evaluation syntax (figure 20); it stores pairs and functions on a heap as indicated by the syntax of figure 20, and aims to enforce type constructors (s, the codomain of $\lfloor \cdot \rfloor$) 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 [83, 86].

Unlike for the higher-order-checking semantics, there is a significant overlap between the Transient rules for typed and untyped redexes. Figure 27 thus presents one notion of reduction. The first group of rules in figure 27 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 handles 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

$\boxed{\text{Transient Syntax}}$ extends First-Order Evaluation Syntax

$$v \;=\; i \mid n \mid \mathsf{p}$$

$\boxed{e; \mathcal{H}; \mathcal{B} \rhd_{\mathsf{T}} e; \mathcal{H}; \mathcal{B}}$ selected rules, omitting error-handling for application and for primitives

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

$\boxed{e; \mathcal{H}; \mathcal{B} \rightarrow_{\mathsf{T}} e; \mathcal{H}; \mathcal{B}}$ is the compatible closure of the relation $\rhd_{\mathsf{T}}$; more precisely:

$$\begin{array}{lll}
\text{if} & e_0; \mathcal{H}_0; \mathcal{B}_0 & \rhd_{\mathsf{T}}\ e_1; \mathcal{H}_1; \mathcal{B}_1 \\
\text{then} & E[e_0]; \mathcal{H}_0; \mathcal{B}_0 & \rightarrow_{\mathsf{T}} E[e_1]; \mathcal{H}_1; \mathcal{B}_1
\end{array}$$

$\boxed{e; \mathcal{H}; \mathcal{B} \rightarrow^{*}_{\mathsf{T}} e; \mathcal{H}; \mathcal{B}}$ is the transitive, reflexive closure of the relation $\rightarrow_{\mathsf{T}}$

Fig. 27. Transient notions of reduction

boundaries associated with the function and its argument.[13] Note that untyped functions may appear in typed contexts and vice-versa because Transient does not create wrappers.

Applications of untyped functions in untyped code do not update the blame map. This allows an implementation to insert checks by rewriting only typed code, leaving untyped code as is. Protected typed code can thus interact with any untyped libraries [86], just like other variants.

Not shown in figure 27 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 blame soundness and completeness. They do, however, preserve the top-level shape of all values in typed code. Blame completeness fails because Transient does not update the blame map when an untyped function is applied in an untyped context.

THEOREM 6.21. *Transient does not satisfy* **TS (1)**.

PROOF SKETCH. Let $e_0 = \mathsf{dyn}\ (\ell_0 \blacktriangleleft (\mathsf{Nat} \Rightarrow \mathsf{Nat}) \blacktriangleleft \ell_1)\ (\lambda x_0. -4)$.

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

but $\nvdash_1 (\lambda x_0. -4) : \mathsf{Nat} \Rightarrow \mathsf{Nat}$. □

THEOREM 6.22. *Transient satisfies* **TS (s)**.

PROOF SKETCH. Recall that **s** maps types to type shapes and the unitype to itself. The proof depends on progress and preservation lemmas for the first-order 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:

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

The check expression guards the context. □

THEOREM 6.23. *Transient does not satisfy* **CM**.

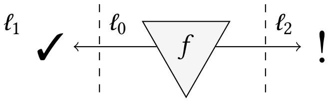
PROOF. A structured value can cross any boundary with a matching shape, regardless of the deeper type structure. For example, the following lifted rule ($\rhd_{\overline{\mathsf{T}}}$) adds a new label to a pair:

$$\begin{aligned}&(\mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \times \tau_1 \blacktriangleleft \ell_1)\ ((\mathsf{p}_0))^{\bar{\ell}_2})^{\ell_3}; \mathcal{H}_0; \mathcal{B}_0 \ \rhd_{\overline{\mathsf{T}}}\ ((\mathsf{p}_0))^{\bar{\ell}_2 \ell_3}; \mathcal{H}_0; \mathcal{B}_1\\ &\quad \text{where } \mathcal{H}_0(\mathsf{p}_0) \in \langle v, v\rangle\end{aligned}$$

□

THEOREM 6.24. *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 ℓ_2 and the irrelevant ℓ_1.



---

[13] Blaming the argument as well as the function is a change to the original Transient semantics [86] that may provide more information in some cases (personal communication with Michael M. Vitousek).

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

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

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

Theorem 6.25. *Transient does not satisfy* **BC**.

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

$$\begin{array}{l}
(\mathsf{app}\{\mathcal{U}\}\ \mathsf{p}_0\ v_0); \mathcal{H}_0; \mathcal{B}_0 \ \ \rhd_{\mathsf{T}}\ \ (e_0[x_0 \leftarrow v_0]); \mathcal{H}_0; \mathcal{B}_0 \\
\quad \text{if } \mathcal{H}_0(\mathsf{p}_0) = \lambda x_0.\, e_0
\end{array}$$

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.

$$\begin{array}{l}
(\mathsf{let}\ f_0 = \mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ (\mathsf{dyn}\ (\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\ (\lambda x_0.\, x_0)^{\ell_2})^{\ell_1}\ \mathsf{in} \\
\ \mathsf{let}\ f_1 = \mathsf{stat}\ (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_3)\ (\mathsf{dyn}\ (\ell_3 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_4)\ (\lambda x_1.\, x_1)^{\ell_4})^{\ell_3}\ \mathsf{in} \\
\ \mathsf{stat}\ (\ell_0 \blacktriangleleft (\mathsf{Int}\times\mathsf{Int}) \blacktriangleleft \ell_5) \\
\qquad (\mathsf{app}\{\mathsf{Int}\times\mathsf{Int}\}\ (\mathsf{dyn}\ (\ell_5 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0)\ (\mathsf{app}\{\mathcal{U}\}\ f_1\ f_0)^{\ell_0})\ 42)^{\ell_5})^{\ell_0}; \emptyset; \emptyset
\end{array}$$

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

Finally, Transient is more permissive than Forgetful in the error pre-order.

Theorem 6.26. $F \lesssim T$.

Proof Sketch. Indirectly, via $T \eqsim A$ (theorem 6.30) and $F \lesssim A$ (theorem 6.31). □

The results about the wrapper-free Transient semantics are negative. It fails **CM** and **BC** because it has no interposition mechanism to keep track of type implications for untyped code. Its heap-based approach to blame fails **BS** because the blame heap conflates different paths in a program.[14]

If several clients use the same library function and one client encounters a type mismatch, every component gets blamed. The reader should keep in mind, however, that the chosen properties are of a purely *theoretical* nature. In *practice*, Transient has played an important role when it comes to performance [33, 36, 37]. Furthermore, the work of Lazarek et al. [45] has also raised questions concerning the pragmatics of blame soundness (and completeness).

6.9 Amnesic and its Properties

The Amnesic semantics (figure 28) employs the same dynamic checks as Transient and supports the synthesis of error messages with path-based blame information. While Transient attempts to track blame with heap addresses, Amnesic uses trace wrappers to attach blame information to values.

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:

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

---

[14] It is possible to adapt the path-based notion of ownership to a form of "shared" ownership that *partially* matches Transient's "collaborative" blame strategy [35]. A notion of ownership that matches Transient fully remains an open problem.

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

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

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

$$\begin{array}{lll}
\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 & \triangleright_{\mathsf{A}} & \mathbb{G}\,(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \\
\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, v_0) & & \\
\quad \text{and } \mathit{rem\text{-}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 & \triangleright_{\mathsf{A}} & v_0 \\
\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, v_0) \text{ and } \mathit{rem\text{-}trace}\,(v_0) \in i & & \\
\mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 & \triangleright_{\mathsf{A}} & \mathsf{BoundaryErr}\,(\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\} \cup b_0^{*}, v_0) \\
\quad \text{if } \neg\mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, v_0) \text{ and } b_0^{*} = \mathit{get\text{-}trace}\,(v_0) & & \\
\mathsf{fst}\{\tau_0\}\;(\mathbb{G}\,(\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)\; v_0) & \triangleright_{\mathsf{A}} & \mathsf{dyn}\; b_0\;(\mathsf{fst}\{\mathcal{U}\}\; v_0) \\
\quad \text{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) & \triangleright_{\mathsf{A}} & \mathsf{dyn}\; b_0\;(\mathsf{snd}\{\mathcal{U}\}\; v_0) \\
\quad \text{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 & \triangleright_{\mathsf{A}} & \mathsf{dyn}\; b_0\;(\mathsf{app}\{\mathcal{U}\}\; v_0\;(\mathsf{stat}\; b_1\; v_1)) \\
\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1) \text{ and } b_1 = (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0) & &
\end{array}$$

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

$$\begin{array}{lll}
\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 & \blacktriangleright_{\mathsf{A}} & \mathbb{G}\,(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \\
\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, v_0) \text{ 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)) & \blacktriangleright_{\mathsf{A}} & \mathsf{trace}\;(\{b_0, b_1\} \cup b_0^{*})\; v_0 \\
\quad \text{if } b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1) \text{ and } \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, v_0) & & \\
\quad \text{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 & \blacktriangleright_{\mathsf{A}} & i_0 \\
\quad \text{if } \mathit{shape\text{-}match}\,(\lfloor \tau_0 \rfloor, i_0) & & \\
\mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 & \blacktriangleright_{\mathsf{A}} & \mathsf{InvariantErr} \\
\quad \text{if } \neg\mathit{shape\text{-}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)) & \blacktriangleright_{\mathsf{A}} & \mathsf{trace}\; b_0^{*}\;(\mathsf{stat}\; b_0\;(\mathsf{fst}\{\tau_0\}\; v_0)) \\
\quad \text{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)) & \blacktriangleright_{\mathsf{A}} & \mathsf{trace}\; b_0^{*}\;(\mathsf{stat}\; b_0\;(\mathsf{snd}\{\tau_1\}\; v_0)) \\
\quad \text{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 & \blacktriangleright_{\mathsf{A}} & \mathsf{trace}\; b_0^{*}\;(\mathsf{stat}\; b_0\;(\mathsf{app}\{\tau_2\}\; v_0\; e_0)) \\
\quad \text{where } \tau_0 = \tau_1 \Rightarrow \tau_2 \text{ and } b_0 = (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1) \text{ and } b_1 = (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0) & & \\
\quad \text{and } e_0 = (\mathsf{dyn}\; b_1\;(\mathit{add\text{-}trace}\,(\mathit{rev}\,(b_0^{*}), v_1))) & & \\
\mathsf{trace}\; b_0^{*}\; v_0 & \blacktriangleright_{\mathsf{A}} & v_1 \\
\quad \text{where } v_1 = \mathit{add\text{-}trace}\,(b_0^{*}, v_0) & &
\end{array}$$

$\boxed{e \rightarrow^{*}_{\mathsf{A}} e}$ is the transitive, reflexive, compatible (with respect to evaluation contexts E, figure 17) closure of the relation $\bigcup\{\triangleright_{\mathsf{A}}, \blacktriangleright_{\mathsf{A}}, \blacktriangleright', \triangleright\}$, where $\blacktriangleright'$ is a variant of $\blacktriangleright$ (figure 22) that calls *rem-trace* on inputs to δ (details in supplement)

Fig. 28. Amnesic notions of reduction

$$\mathit{add\text{-}trace}\,(b_0^*, v_0) = \begin{cases} v_0 \\ \qquad \text{if } b_0^* = \emptyset \\ \mathbb{T}\,(b_0^* \cup b_1^*)\ v_1 \\ \qquad \text{if } v_0 = \mathbb{T}\,b_1^*\,v_1 \\ \mathbb{T}\,b_0^*\,v_0 \\ \qquad \text{if } v_0 \notin \mathbb{T}\,b^*\,v \text{ and } b_0^* \neq \emptyset \end{cases}$$

$$\mathit{get\text{-}trace}\,(v_0) = \begin{cases} b_0^* & \text{if } v_0 = \mathbb{T}\,b_0^*\,v_1 \\ \emptyset & \text{if } v_0 \notin \mathbb{T}\,b^*\,v \end{cases}$$

$$\mathit{rem\text{-}trace}\,(v_0) = \begin{cases} v_1 & \text{if } v_0 = \mathbb{T}\,b_0^*\,v_1 \\ v_0 & \text{if } v_0 \notin \mathbb{T}\,b^*\,v \end{cases}$$

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

Fig. 29. Metafunctions for Amnesic

- At elimination forms, Amnesic checks only the context's type annotation. If 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 successfully whereas Forgetful raises a boundary error.

The elimination rules for guarded pairs show the clearest difference between checks in Amnesic (which mimics Transient) and Forgetful. Amnesic ignores the type in the guard. Forgetful ignores the type annotation on the pair projection.

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

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

Figure 29 defines three metafunctions and one abbreviation for trace wrappers. The metafunctions extend, retrieve, and remove the boundaries associated with a value. The abbreviation simplifies the formulation of the reduction rules as they now 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, because 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 6.27. *Amnesic satisfies* **TS (1)**.

Proof Sketch. By progress and preservation lemmas for the higher-order typing judgment ($\vdash_1$). Amnesic creates and drops wrappers in the same manner as Forgetful (theorem 6.15), 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), which is enough to satisfy type soundness. □

Theorem 6.28. *Amnesic does not satisfy* **CM**.

$\boxed{\mathcal{L}; \ell \Vdash_p e}$ extends $\mathcal{L}; \ell \Vdash e$ to check the labels on trace wrappers

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

Fig. 30. Path-based ownership consistency for trace wrappers

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

$$\begin{array}{l}(\mathsf{stat}\ (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\ (\!(\mathbb{G}\, b_1\ (\!(\mathbb{T}\, b_0^*\ (\!(\lambda x_0.\, x_0)\!)^{\overline{\ell}_2})\!)^{\overline{\ell}_3})\!)^{\overline{\ell}_4}\ )^{\ell_5} \quad \blacktriangleright_{\overline{\mathsf{A}}} \\ \quad (\!(\mathsf{trace}\ (\{(\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1), b_1\} \cup b_0^*)\ (\!(\lambda x_0.\, x_0)\!)^{\overline{\ell}_2})\!)^{\overline{\ell}_3 \overline{\ell}_4 \ell_5}\end{array}$$

□

Theorem 6.29. *Amnesic satisfies* **BS** *and* **BC**.

Proof Sketch. 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 replaces only the rules for trace wrappers (shown in figure 30) and trace expressions. Now consider the guard-dropping rule:

$$\begin{array}{l}(\mathsf{stat}\ (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\ (\!(\mathbb{G}\, b_1\ (\!(\mathbb{T}\, b_0^*\ (\!(\lambda x_0.\, x_0)\!)^{\overline{\ell}_2})\!)^{\overline{\ell}_3})\!)^{\overline{\ell}_4}\ )^{\ell_5} \quad \blacktriangleright_{\overline{\mathsf{A}}} \\ \quad (\!(\mathsf{trace}\ (\{(\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1), b_1\} \cup b_0^*)\ (\!(\lambda x_0.\, x_0)\!)^{\overline{\ell}_2})\!)^{\overline{\ell}_3 \overline{\ell}_4 \ell_5}\end{array}$$

Path-consistency for the redex implies that $\overline{\ell}_3$ and $\overline{\ell}_4$ 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 6.30. $T \approx A$.

Proof Sketch. 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 check the result of elimination forms. In fact, the two compute equivalent results up to wrappers and blame. □

Theorem 6.31. $F \lesssim A$.

Proof Sketch. 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. □

Theorem 6.32. $A \not\lesssim F$.

Proof Sketch. In the following $A \not\lesssim F$ 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 Int, Amnesic reduces the expression to a value. Forgetful detects an error. □

$\boxed{\text{Erasure Syntax}}$ 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}{lll}
\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_0) \\
\quad \text{if } \delta(\mathit{binop}, v_0, v_1) \text{ is undefined and } v_0 \notin i & & \\
\mathit{binop}\{\tau_0\}\ v_0\ v_1 & \rhd_{\mathsf{E}} & \mathsf{BoundaryErr}\ (\emptyset, v_1) \\
\quad \text{if } \delta(\mathit{binop}, v_0, v_1) \text{ is undefined and } v_0 \in i \text{ and } v_1 \notin i & & \\
\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}
$$

$\boxed{e \rightarrow^{*}_{\mathsf{E}} e}$ is the transitive, reflexive, compatible (with respect to evaluation contexts E, figure 17) closure of the relation $\rhd_{\mathsf{E}}$

Fig. 31. Erasure notions of reduction

6.10 Erasure and its Properties

Figure 31 presents the values and notions of reduction for the Erasure semantics. Erasure ignores all types at run-time. 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 tag error. In typed code, 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 to point back to.

Theorem 6.33. *Erasure satisfies neither* **TS (1)** *nor* **TS (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 6.34. *Erasure satisfies* **TS (0)**.

Proof Sketch. By progress and preservation lemmas for the erased "dynamic-typing" judgment ($\vdash_0$). Given well-formed input, every $\rhd_{\mathsf{E}}$ rule yields a dynamically-typed result. □

Theorem 6.35. *Erasure does not satisfy* **CM**.

Proof Sketch. This static-to-dynamic transition $(\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ (v_0)^{\ell_2})^{\ell_3} \rhd_{\bar{\mathsf{E}}} ((v_0))^{\ell_2 \ell_3}$ adds multiple labels to a value. □

Theorem 6.36.

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

Proof Sketch. 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 6.37. *$A \lesssim E$.*

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

Theorem 6.38. *$E \not\lesssim A$.*

Proof Sketch. As a counterexample showing $E \not\lesssim A$, the following term 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 RELATED WORK

Several authors have used cast calculi to design and analyze variants of the Natural semantics. The original work in this lineage is Henglein's coercion calculus [40]. Siek et al. [67] discover several variants by studying two design choices: laziness in higher-order casts and blame-assignment strategies for the dynamic type. Siek et al. [63] present two space-efficient calculi and prove them equivalent to a Natural blame calculus. Siek and Chen [65] generalize these calculi with a parameterized framework and directly model six of them.

The literature has many other variants of the Natural semantics. Some of these are eager, such as AGT [28] and monotonic [64]; others are lazy like Co-Natural [20, 21, 27]. All can be positioned relative to one another by our error preorder.

The KafKa framework expresses all four type-enforcement strategies compared in section 2: Natural (Behavioral), Erasure (Optional) Transient, and Concrete [18]. It thus enables direct comparisons of example programs. The framework is mechanized and has a close correspondence to practical implementations because each type-enforcement strategy is realized as a compiler to a common core language. KafKa does not, however, include a meta-theoretical analysis.

New et al. [53, 54] develop an axiomatic theory of term precision to formalize the gradual guarantee and subsequently derive an order-theoretic specification of casts. This specification of casts is a guideline for how to enforce types in a way that preserves standard type-based reasoning principles. Only the Natural strategy satisfies the axioms.

8 DISCUSSION

One central design issue for languages that can mix typed and untyped code is the semantics of types and specifically how their integrity is enforced as values flow from typed to untyped code and back. Among other things, the choice determines whether static types can be trusted and whether error messages come with useful information when an interaction goes wrong. The first helps the compiler with type-based optimization and influences how a programmer thinks about performance. The second might play a key role when programmers must debug mismatches between types and code. 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 [8, 16, 58]; such properties are a starting point, but make no contribution to the overall goal.

As our analysis 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 require functional APIs[15] or support from the host runtime system. A language without wrappers of any sort sets up weak guarantees by rewriting typed code.

Technically speaking, the paper presents six distinct semantics from four different angles (table 2) and establishes an error preorder relation:

- Type soundness is a relatively weak property; 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.
- Complete monitoring is a stronger property, adapted from the literature on higher-order contracts [23]. It holds when *untyped* code can trust type specifications and vice-versa.

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

- Blame soundness ensures that every boundary in a blame message is potentially responsible. Four strategies satisfy blame soundness relative to a path-based notion of responsibility. Transient fails to satisfies blame soundness because it merges blame information for distinct references to a heap-allocated value (theorem 6.24). Erasure is trivially blame-sound because it gives the programmer zero information.
- Blame completeness ensures that every blame error comes with an overapproximation of the responsible parties. Three of the blame-sound semantics satisfy blame completeness, and Forgetful can be made complete with a straightforward modification. The Erasure strategy trivially fails blame completeness. The Transient strategy fails because it has no way to supervise untyped values that flow through a typed context.

Transient and Erasure provide the weakest guarantees, but they also have a strength that table 2 does not bring across; namely, they are the only strategies that do not require wrapper values. Wrappers impose space costs and time costs; they also raise object identity issues [26, 43, 72, 84]. A wrapper-free strategy with stronger guarantees would therefore be promising. A related topic for future work is to test whether the weaker guarantees of wrapper-free strategies are sufficiently useful in practice. Lazarek et al. [45] find that the gap between Natural blame and Transient blame is smaller than expected across thousands of simulated debugging scenarios. It remains to be seen whether this small gap nevertheless has large implications for working programmers.

The choice of semantics of type enforcement has implications for two major aspects of language design: the performance of an implementation and its acceptance by working developers. Greenman et al. [38] developed an evaluation framework for the performance concern that is slowly gaining in acceptance; Tunnell Wilson et al. [82] present rather preliminary results concerning the acceptance

---

[15] A language with first-class functions can always use lambda as a wrapper [70].

Table 2. Technical contributions

	Natural		Co-Natural		Forgetful		Transient		Amnesic		Erasure
type soundness	**1**		**1**		**1**		**s**		**1**		**0**
complete monitoring	✓		✓		×		×		×		×
blame soundness	✓		✓		✓		×		✓		∅
blame completeness	✓		✓		×†		×		✓		×
error preorder	N	≲	C	≲	F	≲	T	≈	A	≲	E

† satisfiable by adding Amnesic-style trace wrappers, see supplement

by programmers. In conclusion, though, much remains to be done before the community can truly claim to understand this multi-faceted design space.

ACKNOWLEDGMENTS

Michael Ballantyne inspired the strategy-oriented comparisons in section 5. Michael M. Vitousek suggested that Transient is not as unsound as it first seems, which led us toward the bisimilar, sound Amnesic semantics. Amal Ahmed, Stephen Chang, and Max New criticized several of our attempts to explain complete monitoring. Max also provided a brief technical description of his dissertation work. NSF grants CCF 1518844, CCF 1763922, CNS 1823244, and CCF 2030859 (to the CRA for the CIFellows project) provided support.

REFERENCES

[1] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for All. In *POPL*. 201–214.

[2] Alexander Aiken, Edward L. Wimmers, and T.K. Lakshman. 1994. Soft Typing with Conditional Types. In *POPL*. 163–173.

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

[4] Deyaaeldeen Almahallawi. 2020. *Towards Efficient Gradual Typing via Monotonic References and Coercions*. Ph.D. Dissertation. Indiana University.

[5] Christopher Anderson and Sophia Drossopoulou. 2003. BabyJ: from Object Based to Class Based Programming via Types. *WOOD* 82, 7 (2003), 53–81.

[6] Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. 2017. Sound Gradual Typing: only Mostly Dead. *PACMPL* 1, OOPSLA (2017), 54:1–54:24.

[7] Jan A. Bergstra and John V. Tucker. 1983. Initial and Final Algebra Semantics for Data Type Specifications: Two Characterization Theorems. *SIAM Journal of Computing* 12, 2 (1983), 366–387.

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

[9] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. 2009. Thorn: Robust, Concurrent, Extensible Scripting on the JVM. In *OOPSLA*. 117–136.

[10] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. 2016. Practical Optional Types for Clojure. In *ESOP*. 68–94.

[11] Gilad Bracha and David Griswold. 1993. Strongtalk: Typechecking Smalltalk in a Production Environment. In *OOPSLA*. 215–230.

[12] Robert Cartwright. 1980. A Constructive Alternative to Data Type Definitions. In *LFP*. 46–55.

[13] Robert Cartwright and Mike Fagan. 1991. Soft Typing. In *PLDI*. 278–292.

[14] Giuseppe Castagna, Guillaume Duboc, Victor Lanvin, and Jeremy G. Siek. 2019. A Space-Efficient Call-by-Value Virtual Machine for Gradual Set-Theoretic Types. In *IFL*. 8:1–8:12.

[15] Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. *PACMPL* 1, ICFP (2017), 41:1–41:28.

[16] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levy. 2017. Fast and Precise Type Checking for JavaScript. *PACMPL* 1, OOPSLA (2017), 56:1–56:30.

[17] Olaf Chitil. 2012. Practical typed lazy contracts. In *ICFP*. 67–76.

[18] Benjamin W. Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. 2018. KafKa: Gradual Typing for Objects. In *ECOOP*. 12:1–12:23.

[19] Dart. 2020. The Dart type system. https://dart.dev/guides/language/type-system Accessed 2020-09-04.

[20] Markus Degen, Peter Thiemann, and Stefan Wehr. 2012. The Interaction of Contracts and Laziness. In *PEPM*. 97–106.

[21] Christos Dimoulas and Matthias Felleisen. 2011. On Contract Satisfaction in a Higher-Order World. *Transactions on Programming Languages and Systems* 33, 5 (2011), 16:1–16:29.

[22] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. 2011. Correct Blame for Contracts: no More Scapegoating. In *POPL*. 215–226.

[23] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *ESOP*. 214–233.

[24] Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. Collapsible Contracts: Fixing a Pathology of Gradual Typing. *PACMPL* 2, OOPSLA (2018), 133:1–133:27.

[25] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *ICFP*. 48–59.

[26] Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. 2004. Semantic Casts: Contracts and Structural Subtyping in a Nominal World. In *ECOOP*. 364–388.

[27] Robert Bruce Findler, Shu-yu Guo, and Anne Rogers. 2007. Lazy Contract Checking for Immutable Data Structures. In *IFL*. 111–128.

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

[29] Isaac Oscar Gariano, Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2019. Which of My Transient Type Checks Are Not (Almost) Free?. In *VMIL*. 58–66.

[30] Michael Greenberg. 2014. Space-Efficient Manifest Contracts. *CoRR* abs/1410.2813 (2014). https://arxiv.org/abs/1410.2813

[31] Michael Greenberg. 2015. Space-Efficient Manifest Contracts. In *POPL*. 181–194.

[32] Michael Greenberg. 2019. The Dynamic Practice and Static Theory of Gradual Typing. In *SNAPL*. 6:1–6:20.

[33] Ben Greenman. 2020. *Deep and Shallow Types*. Ph.D. Dissertation. Northeastern University.

[34] Ben Greenman and Matthias Felleisen. 2018. A Spectrum of Type Soundness and Performance. *PACMPL* 2, ICFP (2018), 71:1–71:32.

[35] Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2019. Complete Monitors for Gradual Types. *PACMPL* 3, OOPSLA (2019), 122:1–122:29.

[36] Ben Greenman, Lukas Lazarek, Christos Dimoulas, and Matthias Felleisen. 2022. A Transient Semantics for Typed Racket. *Programming* 6, 2 (2022), 1–25.

[37] Ben Greenman and Zeina Migeed. 2018. On the Cost of Type-Tag Soundness. In *PEPM*. 30–39.

[38] Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. 2019. How to Evaluate the Performance of Gradual Type Systems. *Journal of Functional Programming* 29, e4 (2019), 1–45.

[39] Hugo Musso Gualandi and Roberto Ierusalimschy. 2018. Pallene: a statically typed companion language for Lua. In *SBLP*. 19–26.

[40] Fritz Henglein. 1994. Dynamic Typing: Syntax and Proof Theory. *Science of Computer Programming* 22, 3 (1994), 197–230.

[41] David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-Efficient Gradual Typing. *Higher-Order and Symbolic Computation* 23, 2 (2010), 167–189.

[42] Ralf Hinze, Johan Jeuring, and Andres Löh. 2006. Typed Contracts for Functional Programming. In *FLOPS*. 208–225.

[43] Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. 2015. Transparent Object Proxies in JavaScript. In *ECOOP*. 149–173.

[44] Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. Toward Efficient Gradual Typing for Structural Types via Coercions. In *PLDI*. 517–532.

[45] Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2021. How to Evaluate Blame for Gradual Types. *PACMPL* 5, ICFP (2021), 68:1–68:29.

[46] Kuang-Chen Lu, Ben Greenman, Carl Meyer, Dino Viehland, Aniket Panse, and Shriram Krishnamurthi. 2023. Gradual Soundness: Lessons from Static Python. *Programming* 7, 1 (2023), 2:1–2:40.

[47] Andre Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. 2015. A Formalization of Typed Lua. In *DLS*. 13–25.

[48] Jacob Matthews and Robert Bruce Findler. 2009. Operational Semantics for Multi-Language Programs. *Transactions on Programming Languages and Systems* 31, 3 (2009), 1–44.

[49] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences* 17, 3 (1978), 348–375.

[50] David A. Moon. 1974. *MACLISP Reference Manual, Revision 0*. Technical Report. MIT Project MAC.

[51] Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. 2016. Extensible Access Control with Authorization Contracts. In *OOPSLA*. 214–233.

[52] Fabian Muehlboeck and Ross Tate. 2017. Sound Gradual Typing is Nominally Alive and Well. *PACMPL* 1, OOPSLA (2017), 56:1–56:30.

[53] Max S. New. 2020. *A Semantic Foundation for Sound Gradual Typing*. Ph.D. Dissertation. Northeastern University.

[54] Max S. New, Daniel R. Licata, and Amal Ahmed. 2019. Gradual Type Theory. *PACMPL* 3, POPL (2019), 15:1–15:31.

[55] Atsushi Ohori and Kazuhiko Kato. 1993. Semantics for Communication Primitives in a Polymorphic Language. In *POPL*. 99–112.

[56] Norman Ramsey. 2008. Embedding an Interpreted Language Using Higher-Order Functions and Types. *Journal of Functional Programming* 21, 6 (2008), 585–615.

[57] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The Ins and Outs of Gradual Type Inference. In *POPL*. 481–494.

[58] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *POPL*. 167–180.

[59] Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. 2013. The Ruby Type Checker. In *SAC*. 1565–1572.

[60] Gregor Richards, Ellen Arteca, and Alexi Turcotte. 2017. The VM Already Knew That: Leveraging Compile-Time Knowledge to Optimize Gradual Typing. *PACMPL* 1, OOPSLA (2017), 55:1–55:27.

[61] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete Types for TypeScript. In *ECOOP*. 76–100.

[62] Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2019. Transient Typechecks are (Almost) Free. In *ECOOP*. 15:1–15:29.

[63] Jeremy Siek, Peter Thiemann, and Philip Wadler. 2015. Blame and Coercion: Together Again for the First Time. In *PLDI*. 425–435.

[64] Jeremy Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015. Monotonic References for Efficient Gradual Typing. In *ESOP*. 432–456.

[65] Jeremy G. Siek and Tianyu Chen. 2021. Parameterized Cast Calculi and Reusable Meta-Theory for Gradually Typed Lambda Calculi. *Journal of Functional Programming* 31 (2021), e30.

[66] Jeremy G. Siek and Ronald Garcia. 2012. Interpretations of the Gradually-Typed Lambda Calculus. In *SFP*. 68–80.

[67] Jeremy G. Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the Design Space of Higher-Order Casts. In *ESOP*. 17–31.

[68] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *SFP. University of Chicago, TR-2006-06*. 81–92.

[69] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *SNAPL*. 274–293.

[70] Guy Lewis Steele, Jr. 1976. *Lambda The Ultimate Declarative*. Technical Report AI Memo 379. MIT.

[71] Guy L. Steele, Jr. 1990. *Common Lisp* (2nd ed.). Digital Press.

[72] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. Chaperones and Impersonators: Run-time Support for Reasonable Interposition. In *OOPSLA*. 943–962.

[73] Nikhil Swamy, Cédric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. 2014. Gradual Typing Embedded Securely in JavaScript. In *POPL*. 425–437.

[74] Asumu Takikawa, Daniel Feltey, Earl Dean, Robert Bruce Findler, Matthew Flatt, Sam Tobin-Hochstadt, and Matthias Felleisen. 2015. Towards Practical Gradual Typing. In *ECOOP*. 4–27.

[75] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *POPL*. 456–468.

[76] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual Typing for First-Class Classes. In *OOPSLA*. 793–810.

[77] Satish Thatte. 1990. Quasi-static Typing. In *POPL*. 367–381.

[78] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: from Scripts to Programs. In *DLS*. 964–974.

[79] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *POPL*. 395–406.

[80] Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical Types for Untyped Languages. In *ICFP*. 117–128.

[81] Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. 2017. Migratory Typing: Ten Years Later. In *SNAPL*. 17:1–17:17.

[82] Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. 2018. The Behavior of Gradual Types: a User Study. In *DLS*. 1–12.

[83] Michael M. Vitousek. 2019. *Gradual Typing for Python, Unguarded.* Ph.D. Dissertation. Indiana University.

[84] Michael M. Vitousek, Andrew Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for python. In *DLS*. 45–56.

[85] Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. 2019. Optimizing and Evaluating Transient Gradual Typing. In *DLS*. 28–41.

[86] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big Types in Little Runtime: Open-World Soundness and Collaborative Blame for Gradual Type Systems. In *POPL*. 762–774.

[87] Philip Wadler. 2015. A Complement to Blame. In *SNAPL*. 309–320.

[88] Philip Wadler and Robert Bruce Findler. 2009. Well-typed Programs Can't be Blamed. In *ESOP*. 1–15.

[89] Mitchell Wand. 1979. Final Algebra Semantics and Data Type Extensions. *Journal of Computer and System Sciences* 19 (1979), 27–44.

[90] Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. 2017. Mixed Messages: Measuring Conformance and Non-Interference in TypeScript. In *ECOOP*. 28:1–28:29.

[91] Andrew K. Wright and Robert Cartwright. 1994. A Practical Soft Type System for Scheme. In *LFP*. 250–262.

[92] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94. First appeared as Technical Report TR160, Rice University, 1991.

[93] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. Integrating Typed and Untyped Code in a Scripting Language. In *POPL*. 377–388.