Complete Monitors for Gradual Types

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

In the context of gradual typing, type soundness guarantees the safety of typed code. When untyped code fails to respect types, a runtime check finds the discrepancy. As for untyped code, type soundness makes no promises; it does not protect untyped code from mistakes in type specifications and unwarranted blame.

To address the asymmetry, this paper adapts complete monitoring from the contract world to gradual typing. Complete monitoring strengthens plain soundness into a guarantee that catches problems with faulty type specifications. Furthermore, a semantics that satisfies complete monitoring can easily pinpoint the conflict between a type specification and a value. For gradual typing systems that fail complete monitoring, the technical framework provides a source-of-truth to assess the quality of blame.

CCS Concepts: • **Software and its engineering → Semantics**; Constraints; Functional languages.

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

ACM Reference Format:
Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2019. Complete Monitors for Gradual Types. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 122 (October 2019), 29 pages. https://doi.org/10.1145/3360548

1 TYPE SOUNDNESS IS NOT ENOUGH

In a mixed-typed language,[1] type soundness guarantees that the runtime system protects typed code from bad interactions with untyped code. Different languages realize this protection in different ways. Some use higher-order contract wrappers [Tobin-Hochstadt and Felleisen 2008]. Others employ first-order checks in typed code [Roberts et al. 2019; Vitousek 2019; Vitousek et al. 2017]. In both cases, the runtime protections discover when typed code has to deal with untyped values that do not match certain type specifications, and with varying degrees of accuracy, the runtime system can assign blame to the broken components.

The question is what protects untyped code from mistakes in types in this setting. After all, a mixed-typed language allows, and indeed encourages, programmers to write untyped code that relies on type specifications describing other untyped libraries. For example, when designers build a statically-typed extension of a dynamically typed language, they often supply a type assignment for the (untyped) base environment by wrapping thin layers around existing untyped code, just as originally proposed [Tobin-Hochstadt and Felleisen 2006]. New untyped code may rely on these types; but, unsurprisingly, these type assignments come with mistakes [Feldthaus and Møller 2014;

[1]A mixed-typed language combines static and dynamic typing. Section 3 explains the relation to gradual typing.

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

2475-1421/2019/10-ART122
https://doi.org/10.1145/3360548

St-Amour and Toronto 2013]. Programmers may wish to have some assurance that these mistakes are discovered before they affect the result of a computation.

Consider an untyped library `UT` that exports the function $g(x)$ = `"42"`. The library `TforUT` imports `UT` and exports g at type $\mathsf{Int} \Rightarrow \mathsf{Int}$. Now imagine a developer who prototypes a new module `M` in the untyped fragment of the language and relies on `TforUT`'s type specification for g. The untyped code may apply g in a context that expects an integer $(g(42) + 42)$ or one that uses a tag check (if $\mathit{is\text{-}zero}(g(42))$ then _ else _). What happens next is up to the semantics of the underlying language: it may discover the faulty type specification; it may trigger a runtime check that blames the untyped code; or it may silently compute a flawed result.

In general, the wrapper and the first-order checking approach implement different guarantees, even though both satisfy type soundness theorems. A wrapper approach protects untyped code from incorrect types, and a first-order approach does not. While Greenman and Felleisen [2018] point out this difference with examples, they do not characterize it. This paper offers an explanation in terms of complete monitoring [Dimoulas et al. 2012]. Our adaptation of this property to a mixed-typed language demands two qualities beyond soundness. First, a language must enforce types with runtime checks for *every* channel of communication between typed and untyped code fragments. Second, these checks must enforce the behaviors allowed by the types. The implementation of complete monitoring demands a mechanism for tracking types, something that is occasionally impossible [Vitousek et al. 2017] and always expensive [Allende et al. 2013; Greenman et al. 2019b; Takikawa et al. 2015]. Studying typed-untyped interaction from the perspective of complete monitoring, though, suggests weaker properties and a compromise.

The paper makes three contributions. First, it adapts the notion of *complete monitoring* to wrapper-based and first-order mixed-typed languages. Second, it uses the technical framework to *assess the quality of blame assignments* in systems that fail to satisfy complete monitoring. Third, it presents an approach to runtime checking, dubbed *Amnesic*, that satisfies the same type soundness as the wrapper approach and discovers the same errors as the first-order approach. This compromise semantics fails complete monitoring but satisfies our blame requirements, and thus demonstrates how this investigation opens a new way of exploring the design space of mixed-typed guarantees.

2 MOTIVATIONAL EXAMPLES, THE BASIC INSIGHTS

Our work focuses on the theories underlying two practical gradual/migratory typing systems: Typed Racket [Tobin-Hochstadt and Felleisen 2008; Tobin-Hochstadt et al. 2017] and Transient Reticulated Python [Vitousek 2019; Vitousek et al. 2017]. Each adds a type system to an existing untyped language, satisfies a non-trivial type soundness theorem, and satisfies graduality [New and Ahmed 2018; Siek et al. 2015b] for simply-typed programs.

Figure 1 displays a Typed Racket program that consists of three modules. The module on the left represents an untyped library. The module in the middle is a typed adapter module; it imports identifiers from the library, specifies their types, and immediately re-exports them. Adapters make libraries accessible to typed code, but untyped clients such as the module on the right can also use them. The creators of such modules are likely to rely on the type specifications; the types in this adaptor, however, are faulty. The purpose of this example is to illustrate how Typed Racket protects untyped code from incorrect type annotations (section 2.1).

Figure 2 is a three-module Reticulated program that uses types in the same manner as the example in figure 1. The purpose of this example is to show how a type-sound first-order approach to runtime checking can yield misleading error messages even when the types are correct (section 2.2).

Complete monitoring explains the difference between the two approaches. Based on this insight, section 2.3 surveys our contributions and section 2.4 motivates the Amnesic semantics.

2.1 What Protects the Untyped Code

Figure 1 represents a highly simplified real-world scenario. The module on the left, `net/url`, is an excerpt from an untyped library that has been part of Racket for two decades. The typed module in the middle defines type annotations for the untyped library. Lastly, the module on the right represents the untyped prototype of a `client` application. It imports the typed library, possibly because the developer intends to switch to Typed Racket eventually.

`net/url`

```
#lang racket

_ _ _ 600 lines of code _ _ _
_ _ _ plus dependencies _ _ _

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

`typed/net/url`

```
#lang typed/racket

(define-type URL _ _ _)

(require/typed/provide
 ;; from this library
 net/url
 ;; import the following:

 [string->url
  (-> String URL)]

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

`client`

```
#lang racket

(require html typed/net/url)

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

;; constants:
(define URL-str
  "https://sr.ht")

(define URL
  (string->url URL-str))
```

Fig. 1. Using Typed Racket to define an API

The `main` function on the right calls the typed function to open a network connection and read HTML. Semantically, the function `call/input-url` flows from `net/url` to the typed module and then to `client`; the call itself sends `main`'s arguments to the untyped library code via the typed module. The application of `call/input-url` clearly relies on the type specification from `typed/net/url`: the first argument is a URL structure, the second a function that accepts a string, and the third a function that maps an input port to an HTML representation. The type declaration in figure 1 is buggy, however; the first callback of `call/input-url` demands a URL, not a string.

Fortunately for the developer, Typed Racket compiles types to contracts and thereby catches the mismatch. Here, the compilation of `typed/net/url` generates a higher-order function 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 discovers the mismatch and blames the boundary between `client` and `typed/net/url`. The blame message says that `net/url` broke the contract on the back-channel from it to `client`, but warns the developer on the last line with "assuming the contract is correct." A quick look confirms that the contract—that is, the type from which the contract is derived—is wrong. Typed Racket is unusual in this regard; other mixed-typed languages assume that "well-typed programs can't be blamed" [Wadler and Findler 2009].

One such language is Transient Reticulated. Its compiler inserts runtime checks to protect typed code against ill-shaped values from untyped code. Technically, these checks compare the constructor of a value to the constructor of the expected type at every typed elimination form. Although this strategy guarantees that typed operations always receive inputs within their domains, it may not discover when untyped code and type annotations clash.

2.2 How Precise Can Blame Be When Gradual Typing Fails to Monitor All Channels

Figure 2 presents an arrangement of three Transient Reticulated modules, similar to the code in figure 1. The module on the left exports a function that retrieves data from a URL; this function accepts several optional and keyword arguments. The typed adapter module in the middle formulates types for one valid use of the function; a client may supply a URL as a string and a timeout as a pair of floats. These types are correct. The rest of this subsection explains via the code on the right what happens when a client module supplies a tuple that contains an integer and a string.

```
requests
_ _ _ +2,000 lines of code _ _

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

```
typed_requests
import requests as r

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

```
client
from typed_requests import get

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

Fig. 2. Using Reticulated to define an API

Reticulated's runtime checks ensure that the typed `get` receives a string and a tuple, but do *not* validate the tuple's contents. Next, these same arguments pass to the untyped `get` function in the `requests` module. When the untyped `get` eventually uses the string `"zero"` as a float, Python's runtime system raises an exception that originates from the `requests` module.

In this example, the programmer is lucky. The call to the typed version of `get` is still visible in the stack trace because Python fails to implement tail calls properly. The presence of typed `get` provides a hint that it might be at fault. If the maintainers of Python ever changed their mind about tail calls, this hint would disappear. Even worse, if the code implemented the channel from `client` to `requests` via `typed_requests` with OO callbacks instead of pairs, the intermediate typed call may not be on the stack no matter how the compiler handles tail calls.

In sum, types in Transient Reticulated do not monitor all channels of communication among modules. Consequently, errors within one module might be due to false type assumptions. Reticulated attempts to address this problem with a global map from heap addresses to type obligations. Based on the complete-monitor analysis framework, the technical part of the paper demonstrates, however, that even this map may provide misleading blame assignments.

2.3 Informal Overview of Results

Our **first contribution** is the introduction of *complete monitors for gradual types*. The starting point is the notion of complete monitoring for contract systems. Roughly speaking, a contract system is complete if (1) it is possible to attach a contract to every channel of communication between program components and (2) the discovery of a contract violation points to a mismatch between the obligations on a channel between components and a value that passes through this channel. A mixed-typed language satisfies complete monitoring if the translation of types into runtime checks monitors every channel of communication between components.

The first-order approach of Transient Reticulated fails to protect all channels. As the example in the preceding section illustrates, it does not protect channels of communication through a pair.

A precise statement of complete monitoring depends on a notion of *ownership* of a value by a component. The notion of ownership in this paper is standard [Dimoulas et al. 2011, 2012]. Technically, a semantics (of a mixed-typed language) satisfies complete monitoring if every reduction step yields a *consistent* ownership assignment in which every expression has a unique owner. When a value traverses a communication channel, the obligations are checked. If all of them can be checked off, ownership is transferred; otherwise the receiving component must share ownership with the sending one. Only a wrapping semantics, such as Typed Racket's, implements the ownership consistency requirement of complete monitoring.

Our **second contribution** is a technique to assess the quality of blame assignments in systems that fail complete monitoring. If a value may have multiple owners and a blame assignment may point to multiple components, there are two possibilities:

- A mixed-typed language's blame system is *sound* if all reported blame labels are a *subset* of the ownership labels of the witness value.
- It is *complete* if all reported blame labels are a *superset* of the ownership labels of this value.

Our **third contribution** is a semantics that satisfies the same type soundness property as Typed Racket but checks the same first-order properties as Transient. This proof-of-concept semantics is dubbed Amnesic; it is inspired by forgetful and heedful contracts [Greenberg 2015].

In sum, our formal results justify the entries in table 1. The rows align with the four identified properties: type soundness, complete monitoring, blame soundness, and blame completeness. The columns represent four semantics: N (Natural) for the Typed Racket approach; T for Transient Reticulated; A for Amnesic; and E for Erasure, the semantics of optional type systems. As the first row shows, Natural and Amnesic satisfy the *same* type soundness property, while the properties for Transient and Erasure differ from those and each other. (The notation is explained in section 5; suffice it to say that **1** promises the strongest guarantees for types and **0** the weakest.) Otherwise, only Typed Racket satisfies all properties while Amnesic is blame-sound and blame-complete.

Table 1. Informal summary

	N	T	A	E
type sound	**1**	$\lfloor\cdot\rfloor$	**1**	**0**
complete monitor	✓	×	×	×
sound blame	✓	×	✓	×
complete blame	✓	×	✓	×

2.4 Informal Overview of the Amnesic Semantics

Our three semantics are based on different strategies for mixing typed and untyped program components at runtime. The Natural semantics strictly enforces the boundaries between typed and untyped code. A value may cross a boundary if all properties can be checked. If not, a *monitor wrapper* provides controlled access to the value. A client may send input to the monitor, which: checks the input, forwards input to the value, and checks the result before returning to the client. This well-known wrapping strategy thus guarantees the required ownership consistency guarantee.

The Transient semantics enforces the outermost constructor of types with shallow *tag* checks in statically-typed code. These checks guard every untyped-to-typed boundary and every typed elimination form (function application, pair projection). If a check succeeds, Transient records the fact in a global blame map from heap addresses to sets of boundaries. If a later check fails, the map provides some information about what may have caused the failure.

The contrast between Natural and Transient is striking. While the former may wrap a value in an unbounded number of proxy layers,[2] the latter uses no wrappers and still protects typed code. The only significant drawback to Transient appears to be the imprecise blame map.

The Amnesic semantics is a compromise between these two extremes. Amnesic performs the same tag checks as Transient. Instead of a global map, though, it attaches blame metadata to values. In our model, a *trace* wrapper records the boundaries that a value has previously crossed. If an untyped function enters a typed component, Amnesic wraps the function in a monitor. If the function travels back to untyped code, Amnesic replaces the monitor 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 monitor to protect its current typed client, a middle trace to record its last N trips, and an inner monitor to protect its body. Thus Amnesic limits the depth of wrappers and tracks relevant blame information.

[2]Proxies may be encoded to save space [Greenberg 2015; Herman et al. 2010; Siek et al. 2015a].

$\boxed{\text{Surface Language}}$

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

$\boxed{e\!:\!\tau?\ \mathbf{wf}}$

$$\begin{array}{l}
e_0\!:\!\tau_0\ \mathbf{wf}\ \text{ iff } \exists\,\ell_0.\, \ell_0 \Vdash e_0 \text{ and } \vdash e_0 : \tau_0 \\
e_0\!:\!\mathcal{U}\ \mathbf{wf}\ \text{ iff } \exists\,\ell_0.\, \ell_0 \Vdash e_0 \text{ and } \vdash e_0 : \mathcal{U}
\end{array}$$

$\boxed{L;\ell \Vdash e}$ (selected rules)

$$\frac{(x_0\!:\!\ell_0) \in L_0}{L_0;\ell_0 \Vdash x_0}
\qquad
\frac{(x_0\!:\!\ell_0), L_0;\ell_0 \Vdash e_0}{L_0;\ell_0 \Vdash \lambda x_0.\, e_0}
\qquad
\frac{(x_0\!:\!\ell_0), L_0;\ell_0 \Vdash e_0}{L_0;\ell_0 \Vdash \lambda(x_0\!:\!\tau_0).\, e_0}
\qquad
\frac{L_0;\ell_1 \Vdash e_0}{L_0;\ell_0 \Vdash \mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; e_0}$$

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

$$\frac{(x_0\!:\!\tau_0), \Gamma_0 \vdash e_0 : \tau_1}{\Gamma_0 \vdash \lambda(x_0\!:\!\tau_0).\, e_0 : \tau_0 \Rightarrow \tau_1}
\qquad
\frac{\Gamma_0 \vdash e_0 : \tau_0 \qquad \tau_0 \leqslant\!: \tau_1}{\Gamma_0 \vdash e_0 : \tau_1}
\qquad
\frac{\Gamma_0 \vdash e_0 : \mathcal{U}}{\Gamma_0 \vdash \mathsf{dyn}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; e_0 : \tau_0}$$

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

$$\frac{(x_0\!:\!\mathcal{U}), \Gamma_0 \vdash e_0 : \mathcal{U}}{\Gamma_0 \vdash \lambda x_0.\, e_0 : \mathcal{U}}
\qquad
\frac{\Gamma_0 \vdash e_0 : \mathcal{U} \qquad \Gamma_0 \vdash e_1 : \mathcal{U}}{\Gamma_0 \vdash \mathsf{app}\{\mathcal{U}\}\; e_0\; e_1 : \mathcal{U}}
\qquad
\frac{\Gamma_0 \vdash e_0 : \tau_0}{\Gamma_0 \vdash \mathsf{stat}\;(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; e_0 : \mathcal{U}}$$

$\boxed{\tau \leqslant\!: \tau}$

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

Fig. 3. Surface Language

3 THE MODEL: SYNTAX AND TYPES

Our model builds on the previous work of Greenman and Felleisen [2018] who, in turn, employ Matthews and Findler [2009]'s multi-language framework for modeling the syntax, types, and semantics of a mixed-typed language. In particular, we interpret one "surface" syntax and mixed type system (figure 3) in three different ways. This section presents the common basis. The set of surface expressions e includes numbers, tuples, and anonymous functions, illustrative of the atomic values, data structures, and higher-order values. Expressions may be combined through function application (app), operator application (*unop* and *binop*), and boundary terms (dyn and stat).

Boundary terms divide a program into named components. Each component is either statically or dynamically typed. A boundary term combines a boundary specification (b, for instance $(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$) and a sender expression. For example, the boundary term $(\mathsf{dyn}\;(\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\;(\lambda x_0.\, x_0))$ embeds a dynamically-typed sender component named ℓ_1, consisting of the function $(\lambda x_0.\, x_0)$, into a statically-typed context named ℓ_0. While Nat seems to disagree with the enclosed function value, the term is grammatically correct and well-typed.

Every function application and operator application comes with the type $\tau?$ of its expected result.[3] In a statically-typed component, any type τ may be acceptable. Within a dynamically-typed component, the mark $\mathcal{U}$ of a un(i)typed value is the only appropriate declaration.

Note that $\mathcal{U}$ is not a dynamic type [Thatte 1990]. Our model is thus directly representative of languages such as Dart 2 (`dart.dev/dart-2`) and Typed Racket, and of the cast calculi employed by many true gradual languages [Cimini and Siek 2017; Siek et al. 2015b].

An expression is well-formed if it uses component names correctly and is well-typed. Figure 3 defines the relevant judgments. The first judgment, $\Vdash$, checks that components are named consistently and that the variables defined in one component are never directly accessed by another component. This judgment applies to both static and dynamic code. Type-correctness uses two judgments: one for static typing ($\Gamma \vdash e : \tau$) and one for dynamic typing ($\Gamma \vdash e : \mathcal{U}$). These judgments depend on one another to handle boundary terms. For example, a stat boundary has type $\mathcal{U}$ if its subexpression has the type in the boundary specification. The static typing judgment depends on a notion of subtyping ($\leqslant$:)—based on a subset relationship between the Nat and Int types—because we firmly embrace the idea that a type system for untyped code must have true union types [Castagna and Lanvin 2017; Tobin-Hochstadt and Felleisen 2010; Tobin-Hochstadt et al. 2017]. Subtyping is incorporated to the typing judgment via a subsumption rule to keep the presentation brief.

4 THE MODEL: THREE SEMANTICS

The three interpretations of the surface syntax, dubbed Natural, Transient, and Amnesic, are defined via three reduction semantics. The first three subsections introduce the *notions of reduction*; the last one generates the compatible closure with respect to evaluation contexts of these relations in a reasonably standard fashion [Barendregt 1981; Felleisen et al. 2009].

The three semantics utilize some common extensions to the surface syntax and common meta functions; see figures 4, 5,[4] and 6. In particular, each recognizes the same errors (Err), uses the same type constructors (K) for tag checks, and assigns the same meaning (δ) to primitive operations.

A program evaluation may signal four kinds of errors, defined in figure 4. First, a dynamic tag error (TagErr $\bullet$) is the outcome of an evaluation that applies an elimination form to a misshaped argument. For example, the first projection of an integer signals such an error. A static tag error (TagErr $\circ$) results from similar applications in typed code, and from any other redex that contradicts the static typing judgment. Intuitively, type soundness eliminates the possibility of such contradictions. Third, a division-by-zero error (DivErr) may be raised by an application of the quotient primitive; quotient is one representative example of the partial primitives in a full language. Lastly, a boundary error (BndryErr (b^*, v)) indicates a type mismatch between two components and comes with both a set of boundaries and a witness value.[5] The error BndryErr $(\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\}, v_0)$, 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.

[3]These annotations serve two purposes: one essential, one convenient. On elimination forms (app, fst, and snd) they are essential for the first-order interpretations, because those must check the output of certain elimination forms in typed components and thus enforce the type required by the context. Binary operations have annotations to conveniently disambiguate statically-typed and dynamically-typed redexes. All these annotations could be inferred from the type derivation of an unannotated surface program.

[4]The judgment $v_0 \in n$ holds when the value v_0 is a member of the set of natural numbers, and similarly for other objects and sets. By convention: a variable without a subscript typically refers to a set, and a term containing a set describes a comprehension. For example, $(\lambda x.\, v) = \{(\lambda x_i.\, v_j) \mid x_i \in x \text{ and } v_j \in v\}$.

[5]A boundary error is comparable to a blame error [Wadler and Findler 2009]; however, a boundary error emphasizes that either an untyped component or a type specification may be at fault. Type specifications—whether written by a programmer or inferred [Chen and Campora 2019]—can have bugs.

$\boxed{\text{Evaluation Language}}$

$$\begin{array}{lcl} \mathsf{Err} & = & \mathsf{TagErr}\,\bullet \mid \mathsf{TagErr}\,\circ \mid \mathsf{DivErr} \mid \mathsf{BndryErr}\,(b^*, v) \\ e & = & \ldots \mid \mathsf{Err} \\ K & = & \mathsf{Nat} \mid \mathsf{Int} \mid \mathsf{Pair} \mid \mathsf{Fun} \end{array}$$

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

$$\frac{}{\Gamma_0 \vdash \mathsf{Err} : \tau_0}$$

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

$$\frac{}{\Gamma_0 \vdash \mathsf{Err} : \mathcal{U}}$$

Fig. 4. Syntax for basic errors and type constuctors, common typing judgments

$$tag\text{-}match(K_0, v_0) = \begin{cases} \mathsf{True} \\ \quad \text{if } K_0 = \mathsf{Nat} \text{ and } v_0 \in n \\ \quad \text{or } K_0 = \mathsf{Int} \text{ and } v_0 \in i \\ \quad \text{or } K_0 = \mathsf{Pair} \text{ and} \\ \qquad v_0 \in \langle v, v\rangle \cup \\ \qquad\qquad (\mathsf{mon}\,(\ell\blacktriangleleft(\tau\times\tau)\blacktriangleleft\ell)\; v) \\ \quad \text{or } K_0 = \mathsf{Fun} \text{ and} \\ \qquad v_0 \in (\lambda x.\, e) \cup (\lambda(x\!:\!\tau).\, e) \cup \\ \qquad\qquad (\mathsf{mon}\,(\ell\blacktriangleleft(\tau\Rightarrow\tau)\blacktriangleleft\ell)\; v) \\ tag\text{-}match(K_0, v_1) \\ \quad \text{if } v_0 = \mathsf{trace_v}\, b_0^*\, v_1 \\ \mathsf{False} \\ \quad \text{otherwise} \end{cases}$$

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

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

Fig. 5. Common metafunctions

$$\delta(unop, \langle v_0, v_1\rangle) = \begin{cases} v_0 \\ \quad \text{if } unop = \mathsf{fst}\{\tau?\} \\ v_1 \\ \quad \text{if } unop = \mathsf{snd}\{\tau?\} \end{cases} \qquad \delta(binop, i_0, i_1) = \begin{cases} i_0 + i_1 \\ \quad \text{if } binop = \mathsf{sum}\{\tau?\} \\ \mathsf{DivErr} \\ \quad \text{if } binop = \mathsf{quotient}\{\tau?\} \\ \quad \text{and } i_1 = 0 \\ \lfloor i_0 / i_1 \rfloor \\ \quad \text{if } binop = \mathsf{quotient}\{\tau?\} \\ \quad \text{and } i_1 \neq 0 \end{cases}$$

Fig. 6. Specification for primitive operations

4.1 Natural **Notions of Reduction**

Figure 7 extends the base grammar of evaluation expressions with monitor wrappers. While the grammar is somewhat liberal, Natural only ever pairs a function-type boundary with a (possibly-monitored) function value in a monitor. Thus the monitors that arise during evaluation are members of the following two mutually-recursive sets:

$$\begin{array}{lcl} stat\text{-}mon & = & \mathsf{mon}\,(\ell\blacktriangleleft(\tau\Rightarrow\tau)\blacktriangleleft\ell)\;\lambda x.\, e \\ & \mid & \mathsf{mon}\,(\ell\blacktriangleleft(\tau\Rightarrow\tau)\blacktriangleleft\ell)\; dyn\text{-}mon \end{array} \qquad \begin{array}{lcl} dyn\text{-}mon & = & \mathsf{mon}\,(\ell\blacktriangleleft(\tau\Rightarrow\tau)\blacktriangleleft\ell)\;\lambda(x\!:\!\tau).\, e \\ & \mid & \mathsf{mon}\,(\ell\blacktriangleleft(\tau\Rightarrow\tau)\blacktriangleleft\ell)\; stat\text{-}mon \end{array}$$

If any other monitor arises, the notions of reduction raise a static tag error. These rules appear in the technical report along with a proof that well-typed expressions never raise static tag errors.

$\boxed{\text{Natural Evaluation Language}}$

$$v = n \mid i \mid \langle v, v\rangle \mid \lambda x.\, e \mid \lambda(x\!:\!\tau).\, e \mid \mathsf{mon}\; b\; v$$
$$e = \ldots \mid \mathsf{mon}\; b\; v$$

$\boxed{\Gamma \vdash_{\mathsf{N}} e : \tau}$ (selected rules) $\boxed{\Gamma \vdash_{\mathsf{N}} e : \mathcal{U}}$ (selected rules)

$$\frac{\Gamma_0 \vdash_{\mathsf{N}} v_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathsf{N}} \mathsf{mon}\; (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\; v_0 : \tau_0 \Rightarrow \tau_1} \qquad \frac{\Gamma_0 \vdash_{\mathsf{N}} v_0 : \tau_0 \Rightarrow \tau_1}{\Gamma_0 \vdash_{\mathsf{N}} \mathsf{mon}\; (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\; v_0 : \mathcal{U}}$$

Fig. 7. Natural language extensions

4.1.1 Natural*, Statically-Typed.* The δ metafunction (figure 6) defines the semantics of the unary and binary operators. If δ is undefined for the argument values, a tag error results:

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

$unop\{\tau_0\}\; v_0 \quad \rhd_{\mathsf{N}} \quad \mathsf{TagErr} \circ$
  if $\delta(unop, v_0)$ is undefined
$unop\{\tau_0\}\; v_0 \quad \rhd_{\mathsf{N}} \quad \delta(unop, v_0)$
  if $\delta(unop, v_0)$ is defined
$binop\{\tau_0\}\; v_0\; v_1 \quad \rhd_{\mathsf{N}} \quad \mathsf{TagErr} \circ$
  if $\delta(binop, v_0, v_1)$ is undefined
$binop\{\tau_0\}\; v_0\; v_1 \quad \rhd_{\mathsf{N}} \quad \delta(binop, v_0, v_1)$
  if $\delta(binop, v_0, v_1)$ is defined

Only a typed function or monitor may be applied to an argument in a statically-typed context. Any other application is a tag error:

$\mathsf{app}\{\tau_0\}\; v_0\; v_1 \quad \rhd_{\mathsf{N}} \quad \mathsf{TagErr} \circ$
  if $v_0 \notin (\lambda(x\!:\!\tau).\, e) \cup (\mathsf{mon}\; b\; v)$

The application of a typed lambda to an argument is standard:

$\mathsf{app}\{\tau_0\}\; (\lambda(x_0\!:\!\tau_1).\, e_0)\; v_0 \quad \rhd_{\mathsf{N}} \quad e_0[x_0 \leftarrow v_0]$

The application of a monitored, untyped function unfolds the monitor proxy into two new boundary terms. One stat boundary protects the typed argument from improper use by the body of the dynamically-typed function; one dyn boundary checks the result:

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

Both boundaries use the type $(\tau_1 \Rightarrow \tau_2)$ from the monitored function and ignore the annotation τ_0 that decorates the application. The annotations are relevant only for Transient and Amnesic.

The remaining four rules define the behavior of dyn boundaries. These rules initially check a dynamically-typed value against a static type using the *tag-match* metafunction. For functions, a successful check entails the creation of a new monitor:

$\mathsf{dyn}\; (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\; v_0 \quad \rhd_{\mathsf{N}} \quad \mathsf{mon}\; (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\; v_0$
  if $\textit{tag-match}(\lfloor \tau_0 \Rightarrow \tau_1 \rfloor, v_0)$

For pairs, Natural creates a new typed pair containing new dyn boundaries. The evaluation of these boundaries validates the elements of the original, untyped pair:

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

For base types, a successful check is a complete proof that the value matches the type:

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

Otherwise, if the *tag-match* check fails, the reduction ends in a type mismatch. The error message reports the current boundary and the incompatible value:

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

4.1.2 Natural*, Dynamically-Typed.* Applications of primitives end in a dynamic tag error if δ is undefined for the given values:

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

$$\begin{array}{lll} \textit{unop}\{\mathcal{U}\}\,v_0 & \blacktriangleright_{\mathsf{N}} & \mathsf{TagErr}\,\bullet \\ \quad \text{if } \delta(\textit{unop},v_0) \text{ is undefined} && \\ \textit{unop}\{\mathcal{U}\}\,v_0 & \blacktriangleright_{\mathsf{N}} & \delta(\textit{unop},v_0) \\ \quad \text{if } \delta(\textit{unop},v_0) \text{ is defined} && \\ \textit{binop}\{\mathcal{U}\}\,v_0\,v_1 & \blacktriangleright_{\mathsf{N}} & \mathsf{TagErr}\,\bullet \\ \quad \text{if } \delta(\textit{binop},v_0,v_1) \text{ is undefined} && \\ \textit{binop}\{\mathcal{U}\}\,v_0\,v_1 & \blacktriangleright_{\mathsf{N}} & \delta(\textit{binop},v_0,v_1) \\ \quad \text{if } \delta(\textit{binop},v_0,v_1) \text{ is defined} && \end{array}$$

Function application follows the tag of the operator: substitution for an untyped function, decomposition for a monitor, and a tag error for anything else:

$$\begin{array}{lll} \mathsf{app}\{\mathcal{U}\}\,v_0\,v_1 & \blacktriangleright_{\mathsf{N}} & \mathsf{TagErr}\,\bullet \\ \quad \text{if } v_0 \notin (\lambda x.\,e)\cup(\mathsf{mon}\,b\,v) && \\ \mathsf{app}\{\mathcal{U}\}\,(\lambda x_0.\,e_0)\,v_0 & \blacktriangleright_{\mathsf{N}} & e_0[x_0\leftarrow v_0] \\ \mathsf{app}\{\mathcal{U}\}\,(\mathsf{mon}\,(\ell_0\blacktriangleleft(\tau_0\Rightarrow\tau_1)\blacktriangleleft\ell_1)\,v_0)\,v_1 & \blacktriangleright_{\mathsf{N}} & \mathsf{stat}\,b_0\,(\mathsf{app}\{\tau_1\}\,v_0\,(\mathsf{dyn}\,b_1\,v_1)) \\ \quad \text{where } b_0=(\ell_0\blacktriangleleft\tau_1\blacktriangleleft\ell_1) \text{ and } b_1=(\ell_1\blacktriangleleft\tau_0\blacktriangleleft\ell_0) && \end{array}$$

In the monitor case, a dyn boundary checks the argument and a stat boundary protects the result.

The rules for stat boundaries protect typed values from untyped contexts. Protection is crucial for typed functions because an untyped context may supply type-incorrect arguments:

$$\begin{array}{l} \mathsf{stat}\,(\ell_0\blacktriangleleft(\tau_0\Rightarrow\tau_1)\blacktriangleleft\ell_1)\,v_0 \;\blacktriangleright_{\mathsf{N}}\; \mathsf{mon}\,(\ell_0\blacktriangleleft(\tau_0\Rightarrow\tau_1)\blacktriangleleft\ell_1)\,v_0 \\ \quad \text{if } \textit{tag-match}\,(\lfloor\tau_0\Rightarrow\tau_1\rfloor,v_0) \end{array}$$

For typed pairs, a traversal protects higher-order members:

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

Base values do not require protection:

$$\begin{array}{l} \mathsf{stat}\,(\ell_0\blacktriangleleft\tau_0\blacktriangleleft\ell_1)\,i_0 \;\blacktriangleright_{\mathsf{N}}\; i_0 \\ \quad \text{if } \textit{tag-match}\,(\lfloor\tau_0\rfloor,i_0) \end{array}$$

Any other combination of type and value indicates a type mismatch within a statically-typed component, and results in a static tag error:

$$\begin{array}{l} \mathsf{stat}\,(\ell_0\blacktriangleleft\tau_0\blacktriangleleft\ell_1)\,v_0 \;\blacktriangleright_{\mathsf{N}}\; \mathsf{TagErr}\,\circ \\ \quad \text{if } \neg\textit{tag-match}\,(\lfloor\tau_0\rfloor,v_0) \end{array}$$

The stat rules do not output a boundary error because a type mismatch in typed code contradicts the static typing judgment, which is meant to be an invariant.

4.2 Transient **Notion of Reduction**

Figure 8 extends the base evaluation language with pre-values, heaps, and tag-check expressions. The goal is to allocate one instance of every function and pair on a value heap. Technically, a value heap $\mathcal{H}$ maps addresses (p) to pre-values (w). A blame heap $\mathcal{B}$ maps addresses to sets of boundaries according to the blame strategy of Transient Reticulated [Vitousek 2019; Vitousek et al. 2017]. A check expression (check $\tau_?$ e p) validates the result of an elimination form: $\tau_?$ is the expected

$$\boxed{\text{Transient Evaluation Language}}$$

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

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

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

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

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

$\boxed{\mathcal{T};\Gamma \vdash_{\mathsf{T}} e : K}$ (selected rules)

$$\frac{\mathcal{T}_0; (x_0\!:\!\mathcal{U}), \Gamma_0 \vdash_{\mathsf{T}} e_0 : \mathcal{U}}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathsf{T}} \lambda x_0.\, e_0 : \mathsf{Fun}} \qquad \frac{\mathcal{T}_0; \Gamma_0 \vdash_{\mathsf{T}} e_0 : K_0}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathsf{T}} \mathsf{check}\, \tau_0\, e_0\, \mathsf{p}_0 : \lfloor \tau_0 \rfloor} \qquad \frac{\mathcal{T}_0; \Gamma_0 \vdash_{\mathsf{T}} e_0 : \mathcal{U}}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathsf{T}} \mathsf{check}\, \tau_0\, e_0\, \mathsf{p}_0 : \lfloor \tau_0 \rfloor}$$

$\boxed{\mathcal{T};\Gamma \vdash_{\mathsf{T}} e : \mathcal{U}}$ (selected rules)

$$\frac{\mathcal{T}_0; (x_0\!:\!\mathcal{U}), \Gamma_0 \vdash_{\mathsf{T}} e_0 : \mathcal{U}}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathsf{T}} \lambda x_0.\, e_0 : \mathcal{U}} \qquad \frac{\mathcal{T}_0; (x_0\!:\!\tau_0), \Gamma_0 \vdash_{\mathsf{T}} e_0 : K_0}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathsf{T}} \lambda(x_0\!:\!\tau_0).\, e_0 : \mathcal{U}} \qquad \frac{(\mathsf{p}_0\!:\!\mathcal{U}) \in \mathcal{T}_0}{\mathcal{T}_0; \Gamma_0 \vdash_{\mathsf{T}} \mathsf{p}_0 : \mathcal{U}}$$

Fig. 8. Transient language extensions and metafunctions

type, e is the result, and p is the address of the previously-eliminated value. The typing judgments $\mathcal{T};\Gamma \vdash_{\mathsf{T}} e : K$ and $\mathcal{T};\Gamma \vdash_{\mathsf{T}} e : \mathcal{U}$ validate type-tags. Both judgments accept all kinds of values and have analogous rules; refer to the technical report for details.

The figure also defines three meta-functions: $\cdot(\cdot)$, $\cdot[\cdot \mapsto \cdot]$, 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.

4.2.1 Transient. The first rule allocates a new heap address for a pre-value:

$\boxed{e;\mathcal{H};\mathcal{B} \vartriangleright_{\mathsf{T}} e;\mathcal{H};\mathcal{B}}$

$$\begin{array}{l}
\mathsf{w}_0;\mathcal{H}_0;\mathcal{B}_0 \;\vartriangleright_{\mathsf{T}}\; \mathsf{p}_0; (\{\mathsf{p}_0 \mapsto \mathsf{w}_0\} \cup \mathcal{H}_0); (\{\mathsf{p}_0 \mapsto \emptyset\} \cup \mathcal{B}_0) \\
\quad \text{where } \mathsf{p}_0 \text{ fresh in } \mathcal{H}_0 \text{ and } \mathcal{B}_0
\end{array}$$

A pair projection extracts a component from a (heap allocated) value. Because projection is an elimination form, the next step is to tag-check the result against the expected type:

$$\begin{array}{l}
(unop\{\tau_0\}\, v_0);\mathcal{H}_0;\mathcal{B}_0 \;\vartriangleright_{\mathsf{T}}\; \mathsf{TagErr}\, \circ;\mathcal{H}_0;\mathcal{B}_0 \\
\quad \text{if } \delta(unop, \mathcal{H}_0(v_0)) \text{ is undefined} \\
(unop\{\mathcal{U}\}\, v_0);\mathcal{H}_0;\mathcal{B}_0 \;\vartriangleright_{\mathsf{T}}\; \mathsf{TagErr}\, \bullet;\mathcal{H}_0;\mathcal{B}_0 \\
\quad \text{if } \delta(unop, \mathcal{H}_0(v_0)) \text{ is undefined} \\
(unop\{\tau?\}\, \mathsf{p}_0);\mathcal{H}_0;\mathcal{B}_0 \;\vartriangleright_{\mathsf{T}}\; (\mathsf{check}\, \tau?\, \delta(unop, \mathcal{H}_0(\mathsf{p}_0))\, \mathsf{p}_0);\mathcal{H}_0;\mathcal{B}_0 \\
\quad \text{if } \delta(unop, \mathcal{H}_0(\mathsf{p}_0)) \text{ is defined}
\end{array}$$

Binary operations yield new integers, and therefore do not require result checks:

$$\begin{array}{l}
(binop\{\tau_0\}\, v_0\, v_1);\mathcal{H}_0;\mathcal{B}_0 \;\vartriangleright_{\mathsf{T}}\; \mathsf{TagErr}\, \circ;\mathcal{H}_0;\mathcal{B}_0 \\
\quad \text{if } \delta(binop, v_0, v_1) \text{ is undefined} \\
(binop\{\mathcal{U}\}\, v_0\, v_1);\mathcal{H}_0;\mathcal{B}_0 \;\vartriangleright_{\mathsf{T}}\; \mathsf{TagErr}\, \bullet;\mathcal{H}_0;\mathcal{B}_0 \\
\quad \text{if } \delta(binop, v_0, v_1) \text{ is undefined} \\
(binop\{\tau?\}\, i_0\, i_1);\mathcal{H}_0;\mathcal{B}_0 \;\vartriangleright_{\mathsf{T}}\; \delta(binop, i_0, i_1);\mathcal{H}_0;\mathcal{B}_0 \\
\quad \text{if } \delta(binop, i_0, i_1) \text{ is defined}
\end{array}$$

The application of a typed function first confirms the tag of the argument value against the domain of the function. If they match, the rule: (1) extends the blame for the argument with reversed boundaries, to record the flow into the function; (2) substitutes the argument into the function body; and (3) guards the result expression with a codomain check for the expected type:

$$\begin{array}{l}(\mathsf{app}\{\tau?\}\ \mathsf{p}_0\ v_0);\mathcal{H}_0;\mathcal{B}_0 \ \rhd_{\mathsf{T}}\ (\mathsf{check}\,\tau?\ e_0[x_0\leftarrow v_0]\ \mathsf{p}_0);\mathcal{H}_0;\mathcal{B}_0[v_0 \cup \mathit{rev}(\mathcal{B}_0(\mathsf{p}_0))] \\ \quad \text{if } \mathcal{H}_0(\mathsf{p}_0) = \lambda(x_0\!:\!\tau_0).\, e_0 \text{ and } \mathit{tag\text{-}match}(\lfloor\tau_0\rfloor, \mathcal{H}_0(v_0))\end{array}$$

If the domain check fails, then Transient reports a boundary error containing v_0 and the boundaries associated with the procedure $\mathcal{H}_0(\mathsf{p}_0)$:

$$\begin{array}{l}(\mathsf{app}\{\tau?\}\ \mathsf{p}_0\ v_0);\mathcal{H}_0;\mathcal{B}_0 \ \rhd_{\mathsf{T}}\ \mathsf{BndryErr}\,(\mathcal{B}_0(\mathsf{p}_0), v_0);\mathcal{H}_0;\mathcal{B}_0 \\ \quad \text{if } \mathcal{H}_0(\mathsf{p}_0) = \lambda(x_0\!:\!\tau_0).\, e_0 \text{ and } \neg\mathit{tag\text{-}match}(\lfloor\tau_0\rfloor, \mathcal{H}_0(v_0))\end{array}$$

The application of an untyped function in a typed context inserts a check that the function computes a value matching the expected type. In anticipation of a possible check error, the rule updates the blame map:

$$\begin{array}{l}(\mathsf{app}\{\tau_0\}\ \mathsf{p}_0\ v_0);\mathcal{H}_0;\mathcal{B}_0 \ \rhd_{\mathsf{T}}\ (\mathsf{check}\,\tau_0\ e_0[x_0\leftarrow v_0]\ \mathsf{p}_0);\mathcal{H}_0;\mathcal{B}_0[v_0 \cup \mathit{rev}(\mathcal{B}_0(\mathsf{p}_0))] \\ \quad \text{if } \mathcal{H}_0(\mathsf{p}_0) = \lambda x_0.\, e_0\end{array}$$

In an untyped context, the reduction merely performs the required substitution:

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

Invalid applications signal a static or dynamic tag error, depending on the context:

$$\begin{array}{l}(\mathsf{app}\{\tau_0\}\ v_0\ v_1);\mathcal{H}_0;\mathcal{B}_0 \ \rhd_{\mathsf{T}}\ \mathsf{TagErr}\,\circ;\mathcal{H}_0;\mathcal{B}_0 \\ \quad \text{if } \mathcal{H}_0(v_0) \notin (\lambda x.\, e) \cup (\lambda(x\!:\!\tau).\, e) \\ (\mathsf{app}\{\mathcal{U}\}\ v_0\ v_1);\mathcal{H}_0;\mathcal{B}_0 \ \rhd_{\mathsf{T}}\ \mathsf{TagErr}\,\bullet;\mathcal{H}_0;\mathcal{B}_0 \\ \quad \text{if } \mathcal{H}_0(v_0) \notin (\lambda x.\, e) \cup (\lambda(x\!:\!\tau).\, e)\end{array}$$

A dyn boundary checks the tag of a value. If successful, the value crosses the boundary and the blame map records the event. Otherwise, the rule reports the current boundary:

$$\begin{array}{l}(\mathsf{dyn}\ (\ell_0\blacktriangleleft \tau_0\blacktriangleleft \ell_1)\ v_0);\mathcal{H}_0;\mathcal{B}_0 \ \rhd_{\mathsf{T}}\ v_0;\mathcal{H}_0;(\mathcal{B}_0[v_0 \cup \{(\ell_0\blacktriangleleft \tau_0\blacktriangleleft \ell_1)\}]) \\ \quad \text{if } \mathit{tag\text{-}match}(\lfloor\tau_0\rfloor, \mathcal{H}_0(v_0)) \\ (\mathsf{dyn}\ (\ell_0\blacktriangleleft \tau_0\blacktriangleleft \ell_1)\ v_0);\mathcal{H}_0;\mathcal{B}_0 \ \rhd_{\mathsf{T}}\ \mathsf{BndryErr}\,(\{(\ell_0\blacktriangleleft \tau_0\blacktriangleleft \ell_1)\}, v_0);\mathcal{H}_0;\mathcal{B}_0 \\ \quad \text{if } \neg\mathit{tag\text{-}match}(\lfloor\tau_0\rfloor, \mathcal{H}_0(v_0))\end{array}$$

A stat boundary must check the tag of a value to guard against incorrect types. If the type is incorrect, evaluation ends in a static tag error.

$$\begin{array}{l}(\mathsf{stat}\ (\ell_0\blacktriangleleft \tau_0\blacktriangleleft \ell_1)\ v_0);\mathcal{H}_0;\mathcal{B}_0 \ \rhd_{\mathsf{T}}\ v_0;\mathcal{H}_0;(\mathcal{B}_0[v_0 \cup \{(\ell_0\blacktriangleleft \tau_0\blacktriangleleft \ell_1)\}]) \\ \quad \text{if } \mathit{tag\text{-}match}(\lfloor\tau_0\rfloor, \mathcal{H}_0(v_0)) \\ (\mathsf{stat}\ (\ell_0\blacktriangleleft \tau_0\blacktriangleleft \ell_1)\ v_0);\mathcal{H}_0;\mathcal{B}_0 \ \rhd_{\mathsf{T}}\ \mathsf{TagErr}\,\circ;\mathcal{H}_0;\mathcal{B}_0 \\ \quad \text{if } \neg\mathit{tag\text{-}match}(\lfloor\tau_0\rfloor, \mathcal{H}_0(v_0))\end{array}$$

The rules for check expressions are similar to those for dyn boundaries, but have additional information about the source of the target value. A check for the dynamic type $\mathcal{U}$ is a no-op:

$$(\mathsf{check}\,\mathcal{U}\ v_0\ \mathsf{p}_0);\mathcal{H}_0;\mathcal{B}_0 \ \rhd_{\mathsf{T}}\ v_0;\mathcal{H}_0;\mathcal{B}_0$$

Any other check expression matches the value against the type. If the check fails, the error reports the boundary information for both the value and the p_0 address because either set may contain the boundary at the root of the issue:

$$\begin{array}{l}(\mathsf{check}\,\tau_0\ v_0\ \mathsf{p}_0);\mathcal{H}_0;\mathcal{B}_0 \ \rhd_{\mathsf{T}}\ v_0;\mathcal{H}_0;(\mathcal{B}_0[v_0 \cup \mathcal{B}_0(\mathsf{p}_0)]) \\ \quad \text{if } \mathit{tag\text{-}match}(\lfloor\tau_0\rfloor, \mathcal{H}_0(v_0)) \\ (\mathsf{check}\,\tau_0\ v_0\ \mathsf{p}_0);\mathcal{H}_0;\mathcal{B}_0 \ \rhd_{\mathsf{T}}\ \mathsf{BndryErr}\,(\mathcal{B}_0(v_0) \cup \mathcal{B}_0(\mathsf{p}_0), v_0);\mathcal{H}_0;\mathcal{B}_0 \\ \quad \text{if } \neg\mathit{tag\text{-}match}(\lfloor\tau_0\rfloor, \mathcal{H}_0(v_0))\end{array}$$

$\boxed{\text{Amnesic Evaluation Language}}$

$$\begin{array}{rcl} v & = & n \mid i \mid \langle v, v\rangle \mid \lambda x.\, e \mid \lambda(x\!:\!\tau).\, e \mid \\ & & \mathsf{mon}\ b\ v \mid \mathsf{trace_v}\ b^*\ v \mid u \\ e & = & \ldots \mid \mathsf{mon}\ b\ v \mid \mathsf{trace_v}\ b^*\ v \mid \\ & & \mathsf{trace}\ b^*\ e \end{array}$$

$$\textit{add-trace}(b^*_0, v_0) = \begin{cases} v_0 \\ \quad \text{if } b^*_0 = \emptyset \\ \mathsf{trace_v}\ (b^*_0 \cup b^*_1)\ v_1 \\ \quad \text{if } v_0 = \mathsf{trace_v}\ b^*_1\ v_1 \\ \mathsf{trace_v}\ b^*_0\ v_0 \\ \quad \text{if } v_0 \notin \mathsf{trace_v}\ b^*\ v \text{ and } b^*_0 \neq \emptyset \end{cases}$$

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

$$\frac{\Gamma_0 \vdash_{\mathsf{A}} v_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathsf{A}} \mathsf{mon}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 : \tau_0}$$

$$\textit{get-trace}(v_0) = \begin{cases} b^*_0 & \text{if } v_0 = \mathsf{trace_v}\ b^*_0\ v_1 \\ \emptyset & \text{if } v_0 \notin \mathsf{trace_v}\ b^*\ v \end{cases}$$

$$\textit{rem-trace}(v_0) = \begin{cases} v_1 & \text{if } v_0 = \mathsf{trace_v}\ b^*_0\ v_1 \\ v_0 & \text{if } v_0 \notin \mathsf{trace_v}\ b^*\ v \end{cases}$$

$\boxed{\Gamma \vdash_{\mathsf{A}} e : \mathcal{U}}$ (selected rules)

$$\frac{\Gamma_0 \vdash_{\mathsf{A}} v_0 : \tau_0}{\Gamma_0 \vdash_{\mathsf{A}} \mathsf{mon}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ v_0 : \mathcal{U}} \qquad \frac{\Gamma_0 \vdash_{\mathsf{A}} v_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathsf{A}} \mathsf{trace_v}\ b^*_0\ v_0 : \mathcal{U}} \qquad \frac{\Gamma_0 \vdash_{\mathsf{A}} e_0 : \mathcal{U}}{\Gamma_0 \vdash_{\mathsf{A}} \mathsf{trace}\ b^*_0\ e_0 : \mathcal{U}}$$

$$(\mathsf{trace}^?_{\mathsf{v}}\ b^*_0\ v_1) = v_0 \iff \textit{rem-trace}(v_0) = v_1 \text{ and } \textit{get-trace}(v_0) = b^*_0$$

Fig. 9. Amnesic language extensions, metafunctions, and $\mathsf{trace}^?_{\mathsf{v}}$ abbreviation

4.3 Amnesic **Notions of Reduction**

Figure 9 adds monitors and trace wrappers to the base evaluation language, along with metafunctions to extend, inspect, and remove the trace history associated with a value. During evaluation, Amnesic limits the number of wrappers on a value by removing monitors when the value flows to an untyped component. An originally-untyped value gets at most one trace and one temporary "outer" monitor. An originally-typed value gets at most two monitors—one permanent "inner" and one temporary "outer"—and one trace. Using the abbreviation $(\mathsf{trace}^?_{\mathsf{v}}\ b^*\ v)$ for an optionally-traced value (bottom of figure 9), these wrapped values match the following grammars during evaluation:

$$\begin{array}{rcl} \textit{stat-wrap} & = & \mathsf{mon}\ b\ (\mathsf{trace}^?_{\mathsf{v}}\ b^*\ \langle v, v\rangle) \\ & \mid & \mathsf{mon}\ b\ (\mathsf{trace}^?_{\mathsf{v}}\ b^*\ \lambda x.\, e) \\ & \mid & \mathsf{mon}\ b\ (\mathsf{trace}^?_{\mathsf{v}}\ b^*\ (\mathsf{mon}\ b\ \langle v, v\rangle)) \\ & \mid & \mathsf{mon}\ b\ (\mathsf{trace}^?_{\mathsf{v}}\ b^*\ (\mathsf{mon}\ b\ \lambda(x\!:\!\tau).\, e)) \end{array} \qquad \begin{array}{rcl} \textit{dyn-wrap} & = & \mathsf{trace_v}\ b^*\ i \\ & \mid & \mathsf{trace_v}\ b^*\ \langle v, v\rangle \\ & \mid & \mathsf{trace_v}\ b^*\ \lambda x.\, e \\ & \mid & \mathsf{trace}^?_{\mathsf{v}}\ b^*\ (\mathsf{mon}\ b\ \langle v, v\rangle) \\ & \mid & \mathsf{trace}^?_{\mathsf{v}}\ b^*\ (\mathsf{mon}\ b\ \lambda(x\!:\!\tau).\, e) \end{array}$$

The number of boundaries in a trace may grow without bound.

4.3.1 Amnesic*, Statically-Typed.* Two groups of rules handle primitive operations. One group applies the δ metafunction:

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

$\textit{unop}\{\tau_0\}\ v_0 \ \rhd_{\mathsf{A}}\ \mathsf{TagErr} \circ$
  if $v_0 \notin (\mathsf{mon}\ (\ell \blacktriangleleft (\tau\times\tau) \blacktriangleleft \ell)\ v)$ and $\delta(\textit{unop}, v_0)$ is undefined
$\textit{unop}\{\tau_0\}\ v_0 \ \rhd_{\mathsf{A}}\ \delta(\textit{unop}, v_0)$
  if $\delta(\textit{unop}, v_0)$ is defined
$\textit{binop}\{\tau_0\}\ v_0\ v_1 \ \rhd_{\mathsf{A}}\ \mathsf{TagErr} \circ$
  if $\delta(\textit{binop}, v_0, v_1)$ is undefined
$\textit{binop}\{\tau_0\}\ v_0\ v_1 \ \rhd_{\mathsf{A}}\ \delta(\textit{binop}, v_0, v_1)$
  if $\delta(\textit{binop}, v_0, v_1)$ is defined

The other group handles monitored pair values:

$$\begin{array}{l}
\mathsf{fst}\{\tau_0\}\,(\mathsf{mon}\,(\ell_0 \blacktriangleleft (\tau_1 \times \tau_2) \blacktriangleleft \ell_1)\; v_0) \;\;\rhd_{\mathsf{A}}\;\; \mathsf{dyn}\,(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\,(\mathsf{fst}\{\mathcal{U}\}\; v_0) \\
\mathsf{snd}\{\tau_0\}\,(\mathsf{mon}\,(\ell_0 \blacktriangleleft (\tau_1 \times \tau_2) \blacktriangleleft \ell_1)\; v_0) \;\;\rhd_{\mathsf{A}}\;\; \mathsf{dyn}\,(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\,(\mathsf{snd}\{\mathcal{U}\}\; v_0)
\end{array}$$

The projection of an element from a monitored pair creates a boundary term to check the untyped value. The new boundary uses the annotation τ_0 from the operator, exactly like Transient. Type τ_0 may be weaker than the corresponding type in the monitor, but this weaker guarantee is all that the context explicitly relies on.

An invalid application in typed code yields a tag error:

$$\begin{array}{l}
\mathsf{app}\{\tau_0\}\; v_0\; v_1 \;\;\rhd_{\mathsf{A}}\;\; \mathsf{TagErr}\circ \\
\quad \text{if } v_0 \notin (\lambda(x:\tau).\, e) \cup (\mathsf{mon}\; b\; v)
\end{array}$$

The application of an un-monitored function proceeds by substitution:

$$\mathsf{app}\{\tau_0\}\,(\lambda(x_0:\tau_1).\, e_0)\; v_0 \;\;\rhd_{\mathsf{A}}\;\; e_0[x_0 \leftarrow v_0]$$

The application of a monitored function unfolds into two boundaries:

$$\begin{array}{l}
\mathsf{app}\{\tau_0\}\,(\mathsf{mon}\,(\ell_0 \blacktriangleleft (\tau_1 \Rightarrow \tau_2) \blacktriangleleft \ell_1)\,(v_0))\; v_1 \;\;\rhd_{\mathsf{A}}\;\; \mathsf{dyn}\; b_0\;(\mathsf{app}\{\mathcal{U}\}\; v_0\;(\mathsf{stat}\; b_1\; v_1)) \\
\quad \text{where } b_0 = (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1) \text{ and } b_1 = (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0)
\end{array}$$

One boundary protects the argument using the domain type τ_1 from the monitor, and the other validates the result using the type annotation τ_0 from the application. As with monitored pairs, the annotation τ_0 may be weaker than the monitor's type τ_2 but suffices for the context.

Lastly, the rules for dyn boundaries demonstrate Amnesic's first-order type-enforcement strategy. If an untyped function or pair reaches a matching boundary, the following rule creates a new monitor without checking the elements of a pair:

$$\begin{array}{l}
\mathsf{dyn}\,(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \;\;\rhd_{\mathsf{A}}\;\; \mathsf{mon}\,(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \\
\quad \text{if } \textit{tag-match}(\lfloor \tau_0 \rfloor, v_0) \text{ and } \textit{rem-trace}(v_0) \in (\lambda x.\, e) \cup \langle v, v\rangle \cup (\mathsf{mon}\; b\; v)
\end{array}$$

Base values are permitted to flow to the client context if they match the boundary type. The rule removes any trace wrapper because the match fully checks the type:

$$\begin{array}{l}
\mathsf{dyn}\,(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\,(\mathsf{trace}^{?}_{\mathsf{v}}\; b_0^*\; i_0) \;\;\rhd_{\mathsf{A}}\;\; i_0 \\
\quad \text{if } \textit{tag-match}(\lfloor \tau_0 \rfloor, i_0)
\end{array}$$

If a value reaches an incompatible boundary, then Amnesic reports both the current boundary and any boundaries in the value's trace:

$$\begin{array}{l}
\mathsf{dyn}\,(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \;\;\rhd_{\mathsf{A}}\;\; \mathsf{BndryErr}\,(\{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\} \cup b_0^*, \textit{rem-trace}(v_0)) \\
\quad \text{if } \neg \textit{tag-match}(\lfloor \tau_0 \rfloor, v_0) \text{ and } b_0^* = \textit{get-trace}(v_0)
\end{array}$$

4.3.2 Amnesic, *Dynamically-Typed.* Trace-wrapped expressions and values can appear (only) in dynamically-typed contexts; thus every rule needs to handle possibly-traced values.

The first rule merges a trace expression onto a value:

$$\mathsf{trace}\; b_0^*\; v_0 \;\;\blacktriangleright_{\mathsf{A}}\;\; \textit{add-trace}(b_0^*, v_0) \qquad \boxed{e \blacktriangleright_{\mathsf{A}} e}$$

For un-monitored pairs, Amnesic applies δ to project an element and propagates the trace boundaries from the pair onto the element:

$$\begin{array}{l}
\textit{unop}\{\mathcal{U}\}\; v_0 \;\;\blacktriangleright_{\mathsf{A}}\;\; \mathsf{TagErr}\,\bullet \\
\quad \text{where } v_1 = \textit{rem-trace}(v_0) \text{ and } v_1 \notin (\mathsf{mon}\,(\ell \blacktriangleleft \tau \times \tau \blacktriangleleft \ell)\; v) \text{ and } \delta(\textit{unop}, v_1) \text{ is undefined} \\
\textit{unop}\{\mathcal{U}\}\; v_0 \;\;\blacktriangleright_{\mathsf{A}}\;\; \textit{add-trace}(\textit{get-trace}(v_0), \delta(\textit{unop}, v_1)) \\
\quad \text{where } v_1 = \textit{rem-trace}(v_0) \text{ and } \delta(\textit{unop}, v_1) \text{ is defined}
\end{array}$$

Binary operations compute a new value, so traces on the inputs are irrelevant to the result:

$$\begin{array}{l} \mathit{binop}\{\mathcal{U}\}\, v_0\, v_1 \;\blacktriangleright_{\mathsf{A}}\; \mathsf{TagErr}\,\bullet \\ \quad \text{if } \delta(\mathit{binop}, \mathit{rem\text{-}trace}(v_0), \mathit{rem\text{-}trace}(v_1)) \text{ is undefined} \\ \mathit{binop}\{\mathcal{U}\}\, v_0\, v_1 \;\blacktriangleright_{\mathsf{A}}\; \delta(\mathit{binop}, v_2, v_3) \\ \quad \text{where } v_2 = \mathit{rem\text{-}trace}(v_0) \text{ and } v_3 = \mathit{rem\text{-}trace}(v_1) \text{ and } \delta(\mathit{binop}, v_2, v_3) \text{ is defined} \end{array}$$

The projection of an element from a monitored pair creates a new boundary that protects the element using the type from the monitor. The trace of the element is extended with the trace (if any) from the pair:

$$\begin{array}{l} \mathsf{fst}\{\mathcal{U}\}\, (\mathsf{trace}^?_{\mathsf{v}}\, b_0^*\, (\mathsf{mon}\; (\ell_0 \blacktriangleleft (\tau_0 \times \tau_1) \blacktriangleleft \ell_1)\; v_0)) \;\blacktriangleright_{\mathsf{A}}\; \mathsf{trace}\, b_0^*\, (\mathsf{stat}\; (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; (\mathsf{fst}\{\tau_0\}\, v_0)) \\ \mathsf{snd}\{\mathcal{U}\}\, (\mathsf{trace}^?_{\mathsf{v}}\, b_0^*\, (\mathsf{mon}\; (\ell_0 \blacktriangleleft (\tau_0 \times \tau_1) \blacktriangleleft \ell_1)\; v_0)) \;\blacktriangleright_{\mathsf{A}}\; \mathsf{trace}\, b_0^*\, (\mathsf{stat}\; (\ell_0 \blacktriangleleft \tau_1 \blacktriangleleft \ell_1)\; (\mathsf{snd}\{\tau_1\}\, v_0)) \end{array}$$

For un-monitored functions, Amnesic propagates the function's trace to both the result and the argument value. The boundaries around the latter are reversed because the argument flows into the function body:

$$\begin{array}{l} \mathsf{app}\{\mathcal{U}\}\, (\mathsf{trace}^?_{\mathsf{v}}\, b_0^*\, (\lambda x_0.\, e_0))\; v_0 \;\blacktriangleright_{\mathsf{A}}\; \mathsf{trace}\, b_0^*\, (e_0[x_0 \leftarrow v_1]) \\ \quad \text{where } v_1 = \mathit{add\text{-}trace}(\mathit{rev}(b_0^*), v_0) \end{array}$$

For monitored functions, the argument aquires the monitor's reversed trace information and is validated against the domain of the monitor's type. The result of a monitored application is protected using the monitor's codomain and its trace is extended.

$$\begin{array}{l} \mathsf{app}\{\mathcal{U}\}\, (\mathsf{trace}^?_{\mathsf{v}}\, b_0^*\, (\mathsf{mon}\; (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0))\; v_1 \;\blacktriangleright_{\mathsf{A}}\; \mathsf{trace}\, b_0^*\, (\mathsf{stat}\; b_0\; (\mathsf{app}\{\tau_2\}\, v_0\, (\mathsf{dyn}\; b_1\; v_2))) \\ \quad \text{where } \tau_0 = \tau_1 \Rightarrow \tau_2 \text{ and } b_0 = (\ell_0 \blacktriangleleft \tau_2 \blacktriangleleft \ell_1) \text{ and } b_1 = (\ell_1 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0) \\ \quad \text{and } v_2 = \mathit{add\text{-}trace}(\mathit{rev}(b_0^*), v_1) \end{array}$$

Applying any other kind of value to an argument results in a tag error:

$$\begin{array}{l} \mathsf{app}\{\mathcal{U}\}\, v_0\, v_1 \;\blacktriangleright_{\mathsf{A}}\; \mathsf{TagErr}\,\bullet \\ \quad \text{if } v_0 \notin (\lambda x.\, e) \cup (\mathsf{mon}\; (\ell \blacktriangleleft (\tau \Rightarrow \tau) \blacktriangleleft \ell)\; v) \end{array}$$

When a statically-typed function or pair flows to an untyped context for the first time, Amnesic creates a new monitor (to protect either the function, or functions within the pair):

$$\begin{array}{l} \mathsf{stat}\; (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \;\blacktriangleright_{\mathsf{A}}\; \mathsf{mon}\; (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \\ \quad \text{if } \mathit{tag\text{-}match}(\lfloor \tau_0 \rfloor, v_0) \text{ and } v_0 \in (\lambda(x\!:\!\tau).\, e) \cup \langle v, v \rangle \end{array}$$

When a monitored value flows to an untyped component, Amnesic replaces the monitor wrapper with a trace wrapper. If the monitored value is an untyped function or pair, then it flows un-monitored back to the untyped client. If the value is a monitored-and-typed function or pair, then it retains its inner monitor:

$$\begin{array}{l} \mathsf{stat}\; (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; (\mathsf{mon}\; b_1\; v_0) \;\blacktriangleright_{\mathsf{A}}\; \mathsf{trace}\; \{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1), b_1\}\; v_0 \\ \quad \text{if } \mathit{tag\text{-}match}(\lfloor \tau_0 \rfloor, (\mathsf{mon}\; b_1\; v_0)) \end{array}$$

Typed base values may flow to untyped clients. Any other stat boundary contradicts static typing:

$$\begin{array}{l} \mathsf{stat}\; (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; i_0 \;\blacktriangleright_{\mathsf{A}}\; i_0 \\ \quad \text{if } \mathit{tag\text{-}match}(\lfloor \tau_0 \rfloor, i_0) \\ \mathsf{stat}\; (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\; v_0 \;\blacktriangleright_{\mathsf{A}}\; \mathsf{TagErr}\,\circ \\ \quad \text{if } \neg\mathit{tag\text{-}match}(\lfloor \tau_0 \rfloor, v_0) \end{array}$$

4.4 From Notions of Reduction to Semantics

Aside from the global heaps in Transient, the standard operations of compatible closure ($\rightarrow_{\mathsf{X}}$) and reflexive-transitive closure ($\rightarrow^*_{\mathsf{X}}$) suffice to define three reduction relations; refer to a standard text for background or the technical report for details:

$$\rightarrow^*_{\mathsf{N}} = \rightarrow^*_{(\rhd_{\mathsf{N}} \cup \blacktriangleright_{\mathsf{N}})} \qquad \rightarrow^*_{\mathsf{T}} = \rightarrow^*_{\rhd\!\!\!\rhd_{\mathsf{T}}} \qquad \rightarrow^*_{\mathsf{A}} = \rightarrow^*_{(\rhd_{\mathsf{A}} \cup \blacktriangleright_{\mathsf{A}})}$$

To enable uniform statements about the reduction relations, let X range over the set $\{N,T,A\}$ and let $e_0 \rightarrow^*_{\mathsf{T}} e_1$ iff there exists maps $\mathcal{H}_1$ and $\mathcal{B}_1$ such that $e_0; \emptyset; \emptyset \rightarrow^*_{\mathsf{T}} e_1; \mathcal{H}_1; \mathcal{B}_1$ is defined. In this manner, the term $e_0 \rightarrow^*_{\mathsf{X}} e_1$ has meaning for each of the three reduction relations. Similarly, we write $\Gamma_0 \vdash_{\mathsf{T}} e_0 : K_0$ rather than $\mathcal{T}_0; \Gamma_0 \vdash_{\mathsf{T}} e_0 : K_0$ when the heap typing is clear from context.

A reduction relation diverges on an expression, written "$e_0 \rightarrow^*_{\mathsf{X}}$ diverges," if there is an infinite chain $e_0 \rightarrow_{\mathsf{X}} e_1 \rightarrow_{\mathsf{X}} \cdots$ of reductions starting from e_0 (or $e_0; \emptyset; \emptyset$ for Transient).

5 TYPE SOUNDNESS

A type soundness theorem relates the type τ_0 of a well-formed expression e_0 to the possible outcomes of evaluation.[6] In particular, if the evaluation of e_0 results in a value v_0, then the surface type τ_0 predicts some properties of v_0 in the evaluation language. For each combination of surface typing ($\vdash$) and evaluation typing ($\vdash_X$ for $X \in \{N,T,A\}$) judgments, the predictive aspect of a type soundness theorem may be expressed as a function F on types.

Definition 5.1 (type soundness). Let F be a function from surface types to evaluation types. A reduction relation $\rightarrow^*_{\mathsf{X}}$ satisfies $\mathbf{TS}(\vdash_{\mathsf{X}}, F)$ iff for all $e_0 : \tau_0$ **wf** one of the following holds:

- $e_0 \rightarrow^*_{\mathsf{X}} v_0$ and $\vdash_{\mathsf{X}} v_0 : F(\tau_0)$
- $e_0 \rightarrow^*_{\mathsf{X}} e_1$ and $e_1 \in \{\mathsf{TagErr}\,\bullet, \mathsf{DivErr}\} \cup \mathsf{BndryErr}\,(b^*, v)$
- $e_0 \rightarrow^*_{\mathsf{X}}$ diverges

By implication, type soundness states evaluation does not reach certain "wrong" states [Milner 1978]: static tag errors ($\mathsf{TagErr}\,\circ$) and irreducible expressions.

To formulate the type soundness theorem for the three semantics, we require two functions on types. The first, $\lfloor\cdot\rfloor$ from figure 5, maps a surface type to its constructor. The second, $\mathbf{1}$, is the identity function on types.

Theorem 5.2 (type soundness).

(1) $\rightarrow^*_{\mathsf{N}}$ *satisfies* $\mathbf{TS}(\vdash_{\mathsf{A}}, \mathbf{1})$

(2) $\rightarrow^*_{\mathsf{T}}$ *satisfies* $\mathbf{TS}(\vdash_{\mathsf{T}}, \lfloor\cdot\rfloor)$

(3) $\rightarrow^*_{\mathsf{A}}$ *satisfies* $\mathbf{TS}(\vdash_{\mathsf{A}}, \mathbf{1})$

Proof Sketch. By three lemmas per semantics: progress, preservation, and that surface typing implies evaluation typing. The key for the first bullet is that the evaluation syntax of Amnesic extends the one of Natural and the type judgment $\vdash_{\mathsf{A}}$ extends $\vdash_{\mathsf{N}}$; hence the proof can verify the theorem for the *same* type judgment. See the technical report for full proofs. □

6 TYPE SOUNDNESS IS *REALLY* NOT ENOUGH

Theorem 5.2 states that Natural and Amnesic satisfy the *same* type soundness property. This may falsely suggest that Natural and Amnesic produce the same results (values, errors) for all programs. But type soundness merely says that a reduction relation maps well-typed programs to well-typed results according to some typing judgment, and nothing more.

Recall that Amnesic uses lazy monitors to protect pairs of values while Natural eagerly checks pairs, so it is straightforward to construct an example that highlights the differences:

[6] The definition considers closed e_0 without loss of generality; to study an open expression, wrap it in a closing context or lambda. By contrast, prior works that employ an "open-world" soundness theorem restrict the syntax of mixed-typed programs [Tobin-Hochstadt and Felleisen 2006; Vitousek et al. 2017].

$$\mathit{define}\ e_0 = \mathsf{quotient}\{\mathsf{Nat}\}\ (\mathsf{fst}\{\mathsf{Nat}\}\ (\mathsf{dyn}\ (\ell_0 \blacktriangleleft \mathsf{Nat}\times\mathsf{Nat} \blacktriangleleft \ell_1)\ \langle 1, -1\rangle))\ 0$$
$$e_0 \rightarrow^*_{\mathsf{N}} \mathsf{BndryErr}\,(\{(\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\}, -1)$$
$$e_0 \rightarrow^*_{\mathsf{A}} \mathsf{DivErr}$$

Generally speaking, Amnesic is closer to Transient in the sense that both recognize the same set of erroneous expressions. Natural identifies strictly more.

Theorem 6.1. *The following statements hold for all well-formed expressions:*

(1) *If* $e_0 \rightarrow^*_{\mathsf{A}} \mathsf{Err}$ *then* $e_0 \rightarrow^*_{\mathsf{N}} \mathsf{Err}$,

(2) $e_0 \rightarrow^*_{\mathsf{A}} \mathsf{Err}$ *if and only if* $e_0 \rightarrow^*_{\mathsf{T}} \mathsf{Err}$

Proof Sketch. By two stuttering simulations. Natural and Amnesic stutter when: Natural traverses a pair, Natural applies a monitored function, or Amnesic projects an element of a monitored pair. Amnesic and Transient may stutter when Amnesic eliminates a monitor or Transient reaches a check expression. The technical report works through the details. □

Even this theorem does not capture the complete differences between the semantics. Subtle differences show up when a value crosses multiple boundaries. Suppose, as in figure 1, an untyped function flows into a typed component and out again to an untyped client. The client can trust the types in Natural, but not in Amnesic:

$$\mathit{define}\ e_1 = \mathsf{app}\{\mathcal{U}\}\ \mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Nat}\Rightarrow\mathsf{Nat} \blacktriangleleft \ell_1)\ (\mathsf{dyn}\ (\ell_1 \blacktriangleleft \mathsf{Nat}\Rightarrow\mathsf{Nat} \blacktriangleleft \ell_2)\ (\lambda x_0.\, \lambda x_1.\, x_0))\ 8$$
$$e_1 \rightarrow^*_{\mathsf{N}} \mathsf{BndryErr}\,(\{(\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\}, (\lambda x_1.\, 8))$$
$$e_1 \rightarrow^*_{\mathsf{A}} \lambda x_1.\, 8$$

Clearly, Amnesic is missing some essential runtime checks that prevent the discovery of problematic value flows. And type soundness—the idea that typed operations manipulate only values of the correct shape—cannot explain this difference.

The remainder of this paper presents a novel framework for characterizing the differences between two mixed-typed semantics that satisfy the same type soundness property but implement different guarantees for different parts of the code. There are two key insights:

(1) The first is to recognize that channels of communication between typed and untyped code dynamically appear and disappear.
(2) The second is that type specifications impose obligations on these channels, and therefore all channels deserve observation.

The key technique is a tracking device for values, dubbed ownership. Intuitively, a component in a program owns the values that it contains. A reduction semantics "moves" values from one component to another across channels with obligations. If the obligations can be discharged, the transfer is complete and the receiving component takes on full responsibility—or ownership. If not, both components co-own. Hence checking for ownership properties becomes a way to state and check for meta-properties of mixed-typed languages.

7 OWNERSHIP: LABELS AND REDUCTIONS

Ownership labels decorate an expression with the names of the currently-responsible components. In the original program, each component owns all of its subexpressions. An expression reduces to a value; if a value crosses a boundary, it gains a label. A value loses a label only when it fully matches a boundary type. This section provides a formal language of ownership and explains:

- how to lift an expression into an ownership-labeled syntax, and
- how to lift a reduction relation to propagate ownership labels in a path-based sense.

$\boxed{\text{Ownership Language}}$

$$\begin{array}{rcl} e & = & \ldots \mid (e)^{\ell} \mid \mathsf{dyn}\ b\ (e)^{\ell} \mid \\ & & \mathsf{stat}\ b\ (e)^{\ell} \mid \mathsf{mon}\ b\ (e)^{\ell} \\ v & = & \ldots \mid (v)^{\ell} \\ \ell & = & \text{countable set} \\ \ell^{*} & = & \mathcal{P}(\ell) \\ \bar{\ell} & = & \mid \ell\bar{\ell} \end{array}$$

$$\mathit{owners}(v_0) = \begin{cases} \{\ell_0\} \cup \mathit{owners}(v_1) & \text{if } v_0 = (v_1)^{\ell_0} \\ \mathit{owners}(v_1) & \text{if } v_0 = \mathsf{trace}_{\mathsf{v}}\ b_0^{*}\ v_1 \\ \{\} & \text{otherwise} \end{cases}$$

$$\mathit{rev}(\ell_0 \ldots \ell_n) = \ell_n \ldots \ell_0$$

$$e_0 = (\!(e_1)\!)^{\bar{\ell}_0} \iff e_0 = (\cdots (e_1)^{\ell_n} \cdots)^{\ell_1}$$

$\boxed{e : \tau?\ \overline{\mathbf{wf}}}$

$(e_0)^{\ell_0} : \tau_0\ \overline{\mathbf{wf}}$ iff $\ell_0 \mathrel{\overline{\Vdash}} (e_0)^{\ell_0}$ and $\mathrel{\overline{\vdash}} (e_0)^{\ell_0} : \tau_0$

$(e_0)^{\ell_0} : \mathcal{U}\ \overline{\mathbf{wf}}$ iff $\ell_0 \mathrel{\overline{\Vdash}} (e_0)^{\ell_0}$ and $\mathrel{\overline{\vdash}} (e_0)^{\ell_0} : \mathcal{U}$

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

$$\frac{\Gamma_0 \mathrel{\overline{\vdash}} e_0 : \tau_0}{\Gamma_0 \mathrel{\overline{\vdash}} (e_0)^{\ell_0} : \tau_0}$$

$\boxed{\Gamma \mathrel{\overline{\vdash}} e : \mathcal{U}}$ (selected rules)

$$\frac{\Gamma_0 \mathrel{\overline{\vdash}} e_0 : \mathcal{U}}{\Gamma_0 \mathrel{\overline{\vdash}} (e_0)^{\ell_0} : \mathcal{U}}$$

$\boxed{L; \ell \mathrel{\overline{\Vdash}} e}$ (selected rules)

$$\frac{L_0; \ell_0 \mathrel{\overline{\Vdash}} e_0}{L_0; \ell_0 \mathrel{\overline{\Vdash}} (e_0)^{\ell_0}} \qquad \frac{(x_0 : \ell_0) \in L_0}{L_0; \ell_0 \mathrel{\overline{\Vdash}} x_0} \qquad \frac{(x_0 : \ell_0), L_0; \ell_0 \mathrel{\overline{\Vdash}} e_0}{L_0; \ell_0 \mathrel{\overline{\Vdash}} \lambda x_0.\, e_0} \qquad \frac{(x_0 : \ell_0), L_0; \ell_0 \mathrel{\overline{\Vdash}} e_0}{L_0; \ell_0 \mathrel{\overline{\Vdash}} \lambda(x_0 : \tau_0).\, e_0}$$

$$\frac{L_0; \ell_0 \mathrel{\overline{\Vdash}} e_0 \quad L_0; \ell_0 \mathrel{\overline{\Vdash}} e_1}{L_0; \ell_0 \mathrel{\overline{\Vdash}} \mathsf{app}\{\tau?\}\ e_0\ e_1} \qquad \frac{L_0; \ell_1 \mathrel{\overline{\Vdash}} e_0}{L_0; \ell_0 \mathrel{\overline{\Vdash}} \mathsf{dyn}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ (e_0)^{\ell_1}} \qquad \frac{L_0; \ell_1 \mathrel{\overline{\Vdash}} e_0}{L_0; \ell_0 \mathrel{\overline{\Vdash}} \mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ (e_0)^{\ell_1}}$$

$$\frac{L_0; \ell_1 \mathrel{\overline{\Vdash}} v_0}{L_0; \ell_0 \mathrel{\overline{\Vdash}} \mathsf{mon}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ (v_0)^{\ell_1}} \qquad \frac{L_0; \ell_0 \mathrel{\overline{\Vdash}} v_0}{L_0; \ell_0 \mathrel{\overline{\Vdash}} \mathsf{trace}_{\mathsf{v}}\ b_0^{*}\ v_0}$$

Fig. 10. Ownership language extensions

7.1 Ownership-labeled Syntax

The addition of ownership labels adds new syntax and one requirement to an evaluation language. Ownership labels ℓ form a new syntactic category, but correspond to component names. Labels annotate expressions as superscripts: $(e)^{\ell}$. The new requirement is that in each boundary term the enclosed expression comes with a label that matches the sender name, e.g., the unlabeled term $(\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ e_0)$ must add the label ℓ_1 to the inner expression.

Initially, only the immediate subexpressions of boundary terms have an explicit label. All other expressions implicitly have the same owner as the context they appear in. Contracting a redex may combine labels from different subexpressions onto the same term. Hence expressions and values may become nested under a sequence of labels. A value that occurs under several distinct ownership labels (including the label of its context) has multiple owners.

Figure 10 defines an extended grammar, metafunctions, and judgments for an ownership-labeled language. The main judgment, $e : \tau?\ \overline{\mathbf{wf}}$, states that a well-formed expression has a top-most owner, contains names and labels that satisfy the $\overline{\Vdash}$ judgment, and is well-typed. The $\overline{\Vdash}$ judgment formalizes an *ownership consistency relation* with respect to a name ℓ for the enclosing component and a map L from variables to component names.[7] Within one component, consistency requires that every labeled expression $(e_0)^{\ell_0}$ matches the enclosing component and that every variable is bound by a function defined in a matching component. Other values and non-boundary expressions satisfy

[7]Transient additionally requires a heap labeling. The details are standard and appear in the technical report.

consistency if their subexpressions do. At the boundaries between components, the client side of the boundary specification must match the enclosing component and the sender-side name must match the required label on the subexpression.

The metafunction *owners* formally defines the owners of a value as the set of labels around an unlabeled value stripped of any trace-wrapper metadata. Monitor wrappers are not stripped because they represent boundaries. Figure 10 lastly overloads the *rev* metafunction to reverse a sequence of labels, and defines the abbreviation $\llparenthesis e \rrparenthesis^{\overline{\ell}}$ to capture labels around an expression. For example, $\llparenthesis 4 \rrparenthesis^{\ell_0 \ell_1}$ is short for $((4)^{\ell_0})^{\ell_1}$ and $\llparenthesis 5 \rrparenthesis^{\overline{\ell}_0}$ matches 5 with $\overline{\ell}_0$ bound to the empty sequence.

7.2 Ownership-labeled Reduction

A notion of reduction for an evaluation language may be systematically lifted to a labeled language in two steps. Each lifted rule must first handle terms with an arbitrary number of ownership labels per expression and must second propagate labels to reflect changes in the expression.

The following laws explain how to propagate labels. Each handles one scenario in which labels may be transferred or dropped and comes with an example reduction rule. No new labels may be created during a reduction:

(1) Every newly-created value is owned by the enclosing context.
    $(\mathsf{sum}\{\mathsf{Nat}\}\ (2)^{\ell_0}\ (2)^{\ell_1})^{\ell_2} \rightarrow (4)^{\ell_2}$
    The result 4 *is a new value introduced by the runtime system via the* δ *metafunction.*

(2) Every value that flows out of a value v_0 acquires the labels of v_0 and the context.
    $(\mathsf{snd}\{\mathcal{U}\}\ \llparenthesis \langle (1)^{\ell_0}, (2)^{\ell_0} \rangle \rrparenthesis^{\ell_1 \ell_2})^{\ell_3} \rightarrow \llparenthesis 2 \rrparenthesis^{\ell_0 \ell_1 \ell_2 \ell_3}$
    The value 2 *flows out of the pair* $\langle 1, 2 \rangle$ *and thereby acquires the labels on the pair.*

(3) Every value that flows into v_0 acquires the label of the context and the reversed labels of v_0.
    $(\mathsf{app}\{\mathcal{U}\}\ \llparenthesis \lambda x_0.\, \mathsf{fst}\{\mathcal{U}\}\ x_0 \rrparenthesis^{\ell_0 \ell_1}\ (\langle 8, 6 \rangle)^{\ell_2})^{\ell_3} \rightarrow (\llparenthesis \mathsf{fst}\{\mathcal{U}\}\ \llparenthesis \langle 8, 6 \rangle \rrparenthesis^{\ell_2 \ell_3 \ell_1 \ell_0} \rrparenthesis^{\ell_0 \ell_1})^{\ell_3}$
    The argument value $\langle 8, 6 \rangle$ *is input to the function. The substituted body* $(\mathsf{fst}\{\mathcal{U}\}\ \langle 8, 6 \rangle)$ *flows out of the function, and by law 2 acquires the function's labels.*

(4) If a base value reaches a boundary with a matching base type, then the value may drop its current labels and cross the boundary as a new value in the new context.
    $(\mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Nat} \blacktriangleleft \ell_1)\ (0)^{\ell_2 \ell_1})^{\ell_0} \rightarrow (0)^{\ell_0}$
    The value 0 *fully matches the type* Nat $(\lfloor \mathsf{Nat} \rfloor = \mathsf{Nat} \wedge \textit{tag-match}(\lfloor \mathsf{Nat} \rfloor, 0))$.

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

(6) Consecutive equal labels may be dropped.
    $\llparenthesis 0 \rrparenthesis^{\ell_0 \ell_0 \ell_1 \ell_0} = (0)^{\ell_0 \ell_1 \ell_0}$

(7) Labels on an error term may be dropped.
    $E[(\mathsf{DivErr})^{\ell_0}] \rightarrow \mathsf{DivErr}$

All told, the laws reflect an algebraic intuition about how values travel across a program. The same intuition is implicit in section 2 and further motivated in prior work on higher-order contracts [Dimoulas et al. 2012]. Note that function application is the only "input" operation in our models; the addition of mutable cells would add another kind of input.

Lifting does not otherwise change a reduction relation, meaning the behavior of any expression remains the same. After all, the purpose of lifting a base semantics to a labeled language is to prove properties (that can be stated in terms of ownership labels) about the base semantics.

7.3 Lifting by Example

As a first application of the laws, let us modify the Natural rule that creates a new monitor for an untyped function that reaches a boundary. The unlabeled rule follows:

$$\begin{array}{l}\mathsf{dyn}\ (\ell_0\blacktriangleleft(\tau_0\Rightarrow\tau_1)\blacktriangleleft\ell_1)\ v_0 \rhd_{\mathsf{N}} \mathsf{mon}\ (\ell_0\blacktriangleleft(\tau_0\Rightarrow\tau_1)\blacktriangleleft\ell_1)\ v_0\\ \quad \text{if } \textit{tag-match}(\lfloor\tau_0\Rightarrow\tau_1\rfloor, v_0)\end{array}$$

Lifting the left-hand side of the rule introduces a label ℓ_3 for the owner of the context and a sequence of owners $\overline{\ell_2}$ around the untyped function:

$$\begin{array}{l}(\mathsf{dyn}\ (\ell_0\blacktriangleleft(\tau_0\Rightarrow\tau_1)\blacktriangleleft\ell_1)\ (\!(v_0)\!)^{\overline{\ell_2}})^{\ell_3} \rhd_{\overline{\mathsf{N}}} \ \ldots\\ \quad \text{if } \textit{tag-match}(\lfloor\tau_0\Rightarrow\tau_1\rfloor, v_0)\end{array}$$

The lifted rule does not require that the names in the boundary specification match the ownership labels. In particular, both $\ell_0 \neq \ell_3$ and $\ell_1 \neq \overline{\ell_2}$ may hold of a labeled redex.

The propagation laws assign the label of the context to the newly-created monitor (law 1):

$$\begin{array}{l}(\mathsf{dyn}\ (\ell_0\blacktriangleleft(\tau_0\Rightarrow\tau_1)\blacktriangleleft\ell_1)\ (\!(v_0)\!)^{\overline{\ell_2}})^{\ell_3} \rhd_{\overline{\mathsf{N}}} (\mathsf{mon}\ (\ell_0\blacktriangleleft(\tau_0\Rightarrow\tau_1)\blacktriangleleft\ell_1)\ (\!(v_0)\!)^{\overline{\ell_2}})^{\ell_3}\\ \quad \text{if } \textit{tag-match}(\lfloor\tau_0\Rightarrow\tau_1\rfloor, v_0)\end{array}$$

No other laws apply. Laws 2 and 3 do not apply because only one value is involved and nothing flows out of it. Laws 4 and 5 do not apply because v_0 does not cross a boundary; rather, the monitor preserves the ℓ_0– ℓ_1 boundary. Law 6 does not apply because the rule does not create or extend a sequence of labels, and law 7 does not apply because there are no errors.

For a second example, the following lifted Amnesic rule projects the first element of a traced and monitored pair in a dynamically-typed context. The left side of the rule introduces one list of owners around the trace wrapper ($\overline{\ell_4}$), another list around the monitor ($\overline{\ell_3}$), one label for the context within the monitor (ℓ_2), and one label for the outer context (ℓ_5). The rule creates a fst application inside the monitor boundary:

$$\begin{array}{l}(\mathsf{fst}\{\mathcal{U}\}\ (\!(\mathsf{trace}^{?}_{\mathsf{v}}\ b_0^*\ (\!(\mathsf{mon}\ (\ell_0\blacktriangleleft(\tau_0\times\tau_1)\blacktriangleleft\ell_1)\ (v_0)^{\ell_2})\!)^{\overline{\ell_3}})\!)^{\overline{\ell_4}})^{\ell_5}\\ \qquad\qquad\qquad \blacktriangleright_{\overline{\mathsf{A}}} (\mathsf{trace}\ b_0^*\ (\!(\mathsf{stat}\ (\ell_0\blacktriangleleft\tau_0\blacktriangleleft\ell_1)\ (\mathsf{fst}\{\tau_1\}\ v_0)^{\ell_2})\!)^{\overline{\ell_3}\overline{\ell_4}})^{\ell_5}\end{array}$$

The new stat boundary mediates between the same two components as the monitor, and therefore has the same ℓ_2 label. Law 1 applies this label to the fst application. Law 2 determines the labels outside the stat expression because this expression is the result of eliminating two wrappers. The other laws do not apply.

Lifted variants of the semantics from section 4 appear in the technical report. Henceforth, the symbol $\longrightarrow^*_{\overline{\mathsf{X}}}$ refers to the lifted variant of the $\rightarrow^*_{\mathsf{X}}$ reduction relation.

8 COMPLETE MONITORING *"Well-monitored types cannot lie"*

A semantics is a complete monitor if it enforces every boundary between two components with a suitable runtime check [Dimoulas et al. 2012]. Ownership and ownership consistency ($\Vdash$, figure 10) are tools to translate this informal idea into a technical one. First, the ownership laws guarantee that a value loses a label only by satisfying an appropriate boundary-type test. Second, the consistency relationship fails to hold for any program execution during which a value acquires more than one owner. If a lifted semantics preserves the ownership consistency relation, it prevents unchecked values from crossing a boundary.

Definition 8.1 (complete monitor). A reduction relation $\rightarrow^*_{\mathsf{X}}$ satisfies **CM** iff for all well-formed expressions $(e_0)^{\ell_0}$ and expressions e_1, a reduction $(e_0)^{\ell_0} \longrightarrow^*_{\overline{\mathsf{X}}} e_1$ implies $\ell_0 \Vdash e_1$.

Theorem 8.2.

(1) $\rightarrow^*_{\mathsf{N}}$ *satisfies* **CM**

(2) $\rightarrow^*_{\mathsf{T}}$ *does not satisfy* **CM**

(3) $\rightarrow^*_{\mathsf{A}}$ *does not satisfy* **CM**

Of our three semantics, only Natural is a complete monitor. The proof follows from a preservation lemma for the ownership consistency relation.

Lemma 8.3. *If* $\Gamma_0 \vdash_{\mathsf{N}} e_0 : \tau?$ *and* $L_0; \ell_0 \Vdash e_0$ *and* $e_0 \longrightarrow_{\overline{\mathsf{N}}} e_1$ *then* $L_0; \ell_0 \Vdash e_1$.

Proof Sketch. For atomic and compound values, checks are complete and ownership transfer is too. Higher-order values are wrapped with monitors and, by law 1, the reduction propagates the context label to the monitor. In the higher-order stat rule, for example, consistency guarantees that the owners inside the boundary match the sender name ($\overline{\ell_2} = \ell_1 \ldots \ell_1$):

$$(\mathsf{stat}\ (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\ (\!(v_0)\!)^{\overline{\ell_2}})^{\ell_3} \ \blacktriangleright_{\overline{\mathsf{N}}}\ (\mathsf{mon}\ (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\ (\!(v_0)\!)^{\overline{\ell_2}})^{\ell_3}$$
$$\text{if } \mathit{tag\text{-}match}(\lfloor \tau_0 \Rightarrow \tau_1 \rfloor, v_0)$$

Since no other laws apply, the creation of this wrapper clearly preserves ownership consistency. The technical report contains a full proof. □

Transient and Amnesic are not complete monitors because there exist well-formed expressions that lead to violations of the ownership consistency relation. One way to reach a violation is to send a typed, higher-order value to an untyped client. To see why, let us examine the implications of the label-propagation laws on a few higher-order stat rules.

Unlike Natural, the Transient stat rule lets a higher-order value into untyped code without a monitor wrapper. By law 5, the value must acquire another owner; no other laws apply:

$$(\mathsf{stat}\ (\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\ (\!(v_0)\!)^{\overline{\ell_2}})^{\ell_3}; \mathcal{H}_0; \mathcal{B}_0 \ \triangleright_{\overline{\mathsf{T}}}\ (\!(v_0)\!)^{\overline{\ell_2}\ell_3}; \mathcal{H}_0; (\mathcal{B}_0[v_0 \cup \{(\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_1)\}])$$
$$\text{if } \mathit{tag\text{-}match}(\lfloor \tau_0 \Rightarrow \tau_1 \rfloor, \mathcal{H}_0(v_0))$$

Unless $\overline{\ell_2}$ equals ℓ_3 (which, in a well-formed program, only happens when $\ell_0 = \ell_1$), this rule outputs a value with two owners.

The Amnesic semantics includes a stat rule that replaces a monitor wrapper with a trace wrapper. In the lifted version, law 1 assigns the context label to the new trace wrapper. Law 2 propagates the monitor's labels onto v_0 because this value flows out of the monitor:

$$(\mathsf{stat}\ (\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)\ (\!(\mathsf{mon}\ b_1\ v_0)\!)^{\overline{\ell_2}})^{\ell_3} \ \blacktriangleright_{\overline{\mathsf{A}}}\ (\mathsf{trace}\ \{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1), b_1\}\ (\!(v_0)\!)^{\overline{\ell_2}})^{\ell_3}$$
$$\text{if } \mathit{tag\text{-}match}(\lfloor \tau_0 \rfloor, (\mathsf{mon}\ b_1\ v_0))$$

The owners of a trace-wrapped value include the labels on both the wrapper and the value because a trace represents metadata. This rule may therefore create a value with an inconsistent ownership. From here, we just need to develop this line of thought into a full example.

Lemma 8.4. *Let* $e_0 = \mathsf{stat}\ (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Int}) \blacktriangleleft \ell_1)\ (\mathsf{dyn}\ (\ell_1 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Int}) \blacktriangleleft \ell_2)\ (\lambda x_0.\, (\mathsf{sum}\{\mathcal{U}\}\ x_0\ 1))^{\ell_2})^{\ell_1}$ *and* $e_1 = (\mathsf{app}\{\mathcal{U}\}\ e_0\ (\lambda x_1.\, 0))^{\ell_0}$*. Then* $e_1 : \mathcal{U}\ \overline{\mathbf{wf}}$ *and the following statements hold:*

- $e_1; \emptyset; \emptyset \longrightarrow^*_{\overline{\mathsf{T}}} e_2; \mathcal{H}_0; \mathcal{B}_0$ *and* $\ell_0 \not\Vdash e_2$
- $e_1 \longrightarrow^*_{\overline{\mathsf{A}}} e_3$ *and* $\ell_0 \not\Vdash e_3$

Proof. The Transient semantics allocates an address, which then crosses two boundaries. By law 5, the address ends up with three owners:

$$\begin{aligned}
&e_1; \emptyset; \emptyset \longrightarrow^*_{\underline{\mathsf{T}}} (\mathsf{app}\{\mathcal{U}\}\, (\!(\mathsf{p}_0)\!)^{\ell_2\ell_1\ell_0}\, (\lambda x_1.\, 0))^{\ell_0}; \mathcal{H}_0; \mathcal{B}_0 \\
&\quad \text{where } \mathcal{H}_0 = \{\mathsf{p}_0 \mapsto \lambda x_0.\, (\mathsf{sum}\{\mathcal{U}\}\, x_0\, 1)\} \\
&\quad \text{and } \mathcal{B}_0 = \{\mathsf{p}_0 \mapsto \{(\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Int}) \blacktriangleleft \ell_1), (\ell_1 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Int}) \blacktriangleleft \ell_2)\}\}
\end{aligned}$$

The Amnesic semantics first reduces e_0 to a trace-wrapped value with three owners:

$$\begin{aligned}
(e_0)^{\ell_0} &\longrightarrow^*_{\underline{\mathsf{A}}} (\mathsf{stat}\, (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Int}) \blacktriangleleft \ell_1)\, (\mathsf{mon}\, (\ell_1 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Int}) \blacktriangleleft \ell_2)\, (\lambda x_0.\, (\mathsf{sum}\{\mathcal{U}\}\, x_0\, 1))^{\ell_2})^{\ell_1})^{\ell_0} \\
&\longrightarrow^*_{\underline{\mathsf{A}}} (\mathsf{trace}_{\mathsf{v}}\, \{(\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Int}) \blacktriangleleft \ell_1), (\ell_1 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Int}) \blacktriangleleft \ell_2)\}\, (\!(\lambda x_0.\, (\mathsf{sum}\{\mathcal{U}\}\, x_0\, 1))\!)^{\ell_2\ell_1})^{\ell_0} \\
&= v_0
\end{aligned}$$

Therefore e_1 reduces to an application of the same value; this term does not satisfy ownership consistency because *owners*$(v_0) = \{\ell_0, \ell_1, \ell_2\}$:

$$e_1 \longrightarrow^*_{\underline{\mathsf{A}}} (\mathsf{app}\{\mathcal{U}\}\, v_0\, (\lambda x_1.\, 0))^{\ell_0} \qquad \square$$

The formal property of complete monitoring has concrete implications for a developer. Take the examples from the above proof. Once we reduce the expression to canonical form in Transient and Amnesic, the result is a tag error (not a boundary error), meaning the developer receives no hint that the domain of the function type was not checked.

9 IMPLICATIONS FOR BLAME

In addition to the difference between Natural on one hand and Transient and Amnesic on the other, ownership also explains the difference between Transient and Amnesic. While Transient reduces a program to an error if and only if Amnesic does, the two errors may contain completely different blame information. Consider the example of a function f that is defined in component ℓ_0 and exported to two unrelated components: ℓ_1 and ℓ_2. Now, suppose that applications of f in ℓ_1 all return well-typed results, but an application in ℓ_2 ends in a type mismatch (figure 11). Amnesic blames the boundary between ℓ_0 and ℓ_2, which is precisely where the miscommunication occurred. Transient blames both boundaries, even though component ℓ_1 had nothing to do with the mismatch.

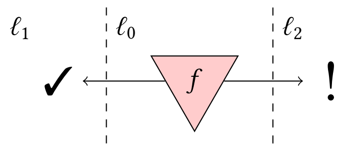

Fig. 11. Function f flows to ℓ_1 and ℓ_2, only ℓ_2 reports a type mismatch

Abstractly, this difference in blame comes about because Amnesic associates boundaries to values using wrappers and Transient combines boundaries in a global map. The question is whether one strategy is better than the other. To decide this question, we need a way to determine which components are responsible for a value.

The concept of ownership provides the ground truth that we need. Let us start from what a boundary error is about. Roughly, a boundary error blames a set of boundaries b_0^* and a value v_0 for a type mismatch. Each boundary specification $(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1)$ in the set is a claim that one component (ℓ_1) sent the value to another component (ℓ_0). If these claims are correct, then a programmer can modify ℓ_1, the boundary type, and/or ℓ_0 to resolve the issue.

Based on this explanation, we can see that blame information can be incorrect in two basic ways. First, the set may contain a boundary that the reported value did not actually cross during evaluation. A blamed set is *unsound* if it includes any such false positives. Second, the set may omit a boundary that the value did in fact cross. A blamed set is *incomplete* if it lacks any part of the value's path through the program.

Using our notion of ownership, we can translate this informal description into a formal one. To do so, we introduce the metafunction *senders*, which returns the component names on the right side of a boundary specifications in a blame set:

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

Now, given a boundary error with its blame set and value, blame soundness and blame completeness compare the senders in the set with the owners of the value.[8]

Definition 9.1 (blame soundness, blame completeness). For all well-formed e_0 such that $e_0 \rightarrow_{\mathsf{X}}^* \mathsf{BndryErr}\,(b_0^*, v_0)$ and, consequently, $(e_0)^{\ell_0} \longrightarrow_{\overline{\mathsf{X}}}^* \mathsf{BndryErr}\,(b_0^*, (\!(v_0)\!)^{\overline{\ell_0}})$:

- $\rightarrow_{\mathsf{X}}^*$ satisfies blame soundness (**BS**) iff $\mathit{senders}(b_0^*) \subseteq \mathit{owners}((\!(v_0)\!)^{\overline{\ell_0}})$
- $\rightarrow_{\mathsf{X}}^*$ satisfies blame completeness (**BC**) iff $\mathit{senders}(b_0^*) \supseteq \mathit{owners}((\!(v_0)\!)^{\overline{\ell_0}})$

Our three semantics relate to blame soundness and completeness in three different ways.

Theorem 9.2.

(1) $\rightarrow_{\mathsf{N}}^*$ *satisfies* **BS** *and* **BC**, *and furthermore blames exactly one boundary*
(2) $\rightarrow_{\mathsf{T}}^*$ *satisfies neither* **BS** *nor* **BC**
(3) $\rightarrow_{\mathsf{A}}^*$ *satisfies* **BS** *and* **BC**

Complete monitoring has major implications for blame soundness and completeness. If a semantics is a complete monitor, then there exists a single-boundary explanation for every type mismatch; namely, the boundary at which the mismatch occurred.

Lemma 9.3. *If* $(e_0)^{\ell_0}$ *is well-formed and* $(e_0)^{\ell_0} \longrightarrow_{\overline{\mathsf{N}}}^* \mathsf{BndryErr}\,(b_0^*, v_0)$, $\mathit{senders}(b_0^*) = \mathit{owners}(v_0)$ *and furthermore* b_0^* *contains one boundary specification.*

Proof. By inspection, the only Natural rule that outputs a boundary error blames a single boundary: $(e_0)^{\ell_0} \longrightarrow_{\overline{\mathsf{N}}}^* E[\mathsf{dyn}\,(\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\,(v_1)^{\ell_2}] \longrightarrow_{\overline{\mathsf{N}}}^* \mathsf{BndryErr}\,(\{(\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\}, (v_1)^{\ell_2})$. Thus $b_0^* = \{(\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\}$ and $\mathit{senders}(b_0^*) = \{\ell_2\}$.
Lemma 8.3 implies $\ell_0 \Vdash E[\mathsf{dyn}\,(\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2)\,(v_1)^{\ell_2}]$, and therefore $\mathit{owners}((v_1)^{\ell_2}) = \{\ell_2\}$. □

The Transient semantics is neither blame-sound nor blame-complete.[9] Blame soundness fails because the global map conflates information when multiple clients reference the same value.

Lemma 9.4. *There exists a well-formed expression* $(e_0)^{\ell_0}$ *such that* $(e_0)^{\ell_0} \longrightarrow_{\overline{\mathsf{T}}}^* \mathsf{BndryErr}\,(b_0^*, v_0)$ *and* $\mathit{senders}(b_0^*) \nsubseteq \mathit{owners}(v_0)$.

Proof. The following example formalizes the illustration in figure 11. Function f_0 enters two unrelated components, causes a type mismatch in one of them, and yet both get blamed.

[8]The properties must consider senders because an error occurs before a value is able to cross an incompatible boundary.
[9]The upcoming Transient counterexamples use let expressions to abbreviate untyped function applications.

$$
\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 \\
\longrightarrow^{*}_{\overline{\mathsf{T}}} (\mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_2)\ (\mathsf{app}\{\mathsf{Int}\}\ (\!(\mathsf{p}_0)\!)^{\ell_0\ell_2}\ 5)^{\ell_2})^{\ell_0}; \mathcal{H}_0; \mathcal{B}_0 \\
\quad \text{where } \mathcal{H}_0 = \{\mathsf{p}_0 \mapsto \lambda x_0.\, \langle x_0, x_0\rangle\} \\
\quad \text{and } \mathcal{B}_0 = \{\mathsf{p}_0 \mapsto (\ell_1 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Int}) \blacktriangleleft \ell_0), (\ell_0 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Int}) \blacktriangleleft \ell_1), (\ell_2 \blacktriangleleft (\mathsf{Int} \Rightarrow \mathsf{Int}) \blacktriangleleft \ell_0)\} \\
\longrightarrow^{*}_{\overline{\mathsf{T}}} (\mathsf{stat}\ (\ell_0 \blacktriangleleft \mathsf{Int} \blacktriangleleft \ell_2)\ (\mathsf{check}\ \mathsf{Int}\ (\!(\mathsf{p}_1)\!)^{\ell_0\ell_2}\ \mathsf{p}_0)^{\ell_2})^{\ell_0}; \mathcal{H}_1; \mathcal{B}_1 \\
\longrightarrow^{*}_{\overline{\mathsf{T}}} \mathsf{BndryErr}\,(\mathcal{B}_1(\mathsf{p}_1) \cup \mathcal{B}_1(\mathsf{p}_0), (\!(\mathsf{p}_1)\!)^{\ell_0\ell_2}); \mathcal{H}_1; \mathcal{B}_1 \\
\quad \text{where } \mathcal{H}_1 = \mathcal{H}_0 \cup \{\mathsf{p}_1 \mapsto \langle 5, 5\rangle\} \\
\quad \text{and } \mathcal{B}_1 = \mathcal{B}_0 \cup \{\mathsf{p}_1 \mapsto \emptyset\}
\end{array}
$$

Thus $\mathit{senders}\,(\mathcal{B}_1(\mathsf{p}_1) \cup \mathcal{B}_1(\mathsf{p}_0)) = \{\ell_0, \ell_1\} \nsubseteq \{\ell_0, \ell_2\} = \mathit{owners}\,((\!(\mathsf{p}_1)\!)^{\ell_0\ell_2})$. Specifically, component ℓ_1 has nothing to do with the type mismatch. □

Blame completeness fails because Transient does not update the blame map when an untyped function is applied in an untyped context. This fact suggests that a mixed-typed language cannot satisfy blame completeness if it has no control over untyped code.

Lemma 9.5. *There exists a well-formed expression* $(e_0)^{\ell_0}$ *such that* $(e_0)^{\ell_0} \longrightarrow^{*}_{\overline{\mathsf{T}}} \mathsf{BndryErr}\,(b^{*}_0, v_0)$ *and* $\mathit{senders}\,(b^{*}_0) \not\supseteq \mathit{owners}\,(v_0)$.

Proof. The expression below presents an untyped function f_1 that updates the type assigned to another function (f_0). Because the update happens in untyped code, the blame map does not record the crucial boundary.

$$
\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))\ \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))\ \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 \\
\longrightarrow^{*}_{\overline{\mathsf{T}}} (\mathsf{stat}\ (\ell_0 \blacktriangleleft (\mathsf{Int}\times\mathsf{Int}) \blacktriangleleft \ell_5) \\
\qquad\quad (\mathsf{app}\{\mathsf{Int}\times\mathsf{Int}\}\ (\mathsf{dyn}\ (\ell_5 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0)\ (\mathsf{app}\{\mathcal{U}\}\ (\!(\mathsf{p}_1)\!)^{\ell_4\ell_3\ell_0}\ (\!(\mathsf{p}_0)\!)^{\ell_2\ell_1\ell_0})^{\ell_0})\ 42)^{\ell_5})^{\ell_0}; \mathcal{H}_0; \mathcal{B}_0 \\
\longrightarrow^{*}_{\overline{\mathsf{T}}} (\mathsf{stat}\ (\ell_0 \blacktriangleleft (\mathsf{Int}\times\mathsf{Int}) \blacktriangleleft \ell_5)\ (\mathsf{app}\{\mathsf{Int}\times\mathsf{Int}\}\ (\!(\mathsf{p}_0)\!)^{\overline{\ell}_0}\ 42)^{\ell_5})^{\ell_0}; \mathcal{H}_1; \mathcal{B}_1 \\
\longrightarrow^{*}_{\overline{\mathsf{T}}} (\mathsf{stat}\ (\ell_0 \blacktriangleleft (\mathsf{Int}\times\mathsf{Int}) \blacktriangleleft \ell_5)\ (\mathsf{check}\ (\mathsf{Int}\times\mathsf{Int})\ (\!(42)\!)^{\overline{\ell}_1}\ \mathsf{p}_0)^{\ell_5})^{\ell_0}; \mathcal{H}_2; \mathcal{B}_2 \\
\longrightarrow^{*}_{\overline{\mathsf{T}}} \mathsf{BndryErr}\,(\mathcal{B}_2(\mathsf{p}_0), (\!(42)\!)^{\overline{\ell}_1}); \mathcal{H}_2; \mathcal{B}_2 \\
\quad \text{where } \tau_0 = (\mathsf{Int} \Rightarrow \mathsf{Int}) \\
\quad \text{and } \tau_1 = (\mathsf{Int} \Rightarrow \mathsf{Int}\times\mathsf{Int}) \\
\quad \text{and } \overline{\ell}_0 = \ell_2\ell_1\ell_0\ell_3\ell_4\ell_3\ell_0\ell_5 \\
\quad \text{and } \overline{\ell}_1 = \ell_5\overline{\ell}_0(\mathit{rev}(\overline{\ell}_0)) \\
\quad \text{and } \mathcal{H}_2 = \{(\mathsf{p}_0 \mapsto \lambda x_0.\, x_0), (\mathsf{p}_1 \mapsto \lambda x_1.\, x_1)\} \\
\quad \text{and } \mathcal{B}_2 = \{(\mathsf{p}_0 \mapsto \{(\ell_0 \blacktriangleleft \tau_0 \blacktriangleleft \ell_1), (\ell_1 \blacktriangleleft \tau_0 \blacktriangleleft \ell_2), (\ell_5 \blacktriangleleft \tau_1 \blacktriangleleft \ell_0)\}), \\
\qquad\qquad\quad (\mathsf{p}_1 \mapsto \{(\ell_0 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_3), (\ell_3 \blacktriangleleft (\tau_0 \Rightarrow \tau_1) \blacktriangleleft \ell_4)\})\}
\end{array}
$$

Hence, $\mathit{senders}\,(\mathcal{B}_2(\mathsf{p}_0)) = \{\ell_0, \ell_1, \ell_2\} \not\supseteq \{\ell_0, \ell_1, \ell_2, \ell_3, \ell_4, \ell_5\} = \mathit{owners}\,((\!(42)\!)^{\overline{\ell}_1})$. The crucial labels ℓ_3 and ℓ_4 are nowhere to be found. □

Finally, to prove blame soundness for Amnesic, we reuse the proof technique for complete monitoring. The key invariant is that only trace-wrapped values may accumulate multiple owners; other terms satisfy the consistency relation. We turn this observation into a judgment, $\overline{\Vdash}_{\mathsf{A}}$, that weakens $\overline{\Vdash}$ just enough. Technically, we weaken the trace rules. For example:

$$\frac{\{\ell_0, \ell_1, \ldots, \ell_n\} = \bigcup\{\{\ell_j, \ell_k\} \mid (\ell_j \blacktriangleleft \tau_0 \blacktriangleleft \ell_k) \in b_0^*\} \qquad L_0; \ell_n \overline{\Vdash}_{\mathsf{A}}\, e_0}{L_0; \ell_0 \overline{\Vdash}_{\mathsf{A}} \left(\mathsf{trace}\, b_0^* \,(\!(e_0)\!)^{\ell_n \cdots \ell_1}\right)^{\ell_0}}$$

Lemma 9.6. *If* $(e_0)^{\ell_0}$ *is well-formed and* $(e_0)^{\ell_0} \longrightarrow^*_{\overline{\mathsf{A}}} \mathsf{BndryErr}\,(b_0^*, v_0)$, *senders*$(b_0^*) =$ *owners*(v_0).

Proof Sketch. By a preservation lemma for the $\overline{\Vdash}_{\mathsf{A}}$ judgment; see the technical report. □

In fact, a variant of Amnesic satisfies a stronger property than lemma 9.6. The variant uses lists of boundaries rather than sets, and a similar proof effort shows that Amnesic reports the inter-component path that propagates a value to a mismatched boundary. We refer the interested reader to the technical report.

9.1 Discussion: The Trade-Off Between the Precision and Cost of Blame

The exploration of blame points to three insights about the precision and cost of blame. First, Natural is best for developers who wish to get precise blame information, and it obviously suffers from high overhead. As Greenman et al. [2019b] point out, the latter is partly due to the wrappers needed to protect typed values from bad untyped code and untyped values from bad types.

Second, the design of Transient represents a rather different trade-off. The runtime cost of types can be much lower than Natural in programs that mix typed and untyped code [Greenman and Felleisen 2018], but Transient types offer few guarantees to programmers. The type soundness theorem for Transient ensures only the top-level shape of values in typed code. Transient fails complete monitoring, meaning it omits some runtime checks to the detriment of the developers of untyped code. And because Transient is neither blame-sound nor blame-complete, it may send developers looking for type violations in the wrong place.

A strong type soundness theorem, complete monitoring, and blame completeness are not in reach for Transient. The design's key axiom is freedom from wrappers [Vitousek et al. 2017], but without wrappers there is no interposition point in untyped code for checking channels and for updating blame information.

There are at least two ways that a variant of Transient can satisfy a notion of blame soundness. One way is to blame fewer boundaries. Reporting zero boundaries in the app and check rules, while keeping the current dyn rules, is sure to eliminate all irrelevant boundaries. A second option is to weaken the ownership rules advocated in section 7. Instead of path-based ownership, a language can advocate a heap-based notion that merges all paths to the same address. It remains to be seen whether heap-based ownership can be useful to developers, but the technical report provides a definition and proves heap-based blame soundness for Transient.

Third, the design of Amnesic is best understood as a variation of Transient that accepts a limited number (three) of wrappers (monitor and/or trace) per value. On one hand, this seemingly small compromise strictly improves the theoretical properties of Transient. On the other hand, Amnesic remains a proof-of-concept semantics.

A practical implementation of Amnesic must address two problems. The first concerns space consumption. Amnesic satisfies blame completeness because it records *every* boundary that a value crosses, meaning this record may actually consume an unbounded amount of space. The simplest fix is to sacrifice blame completeness by limiting the number of elements in a blame set. Another option is to invent a compression scheme that reduces the space needs of a blame set. The second problem concerns the naïve assumption of the model that every value can be wrapped with metadata to trace its provenance. This is clearly impractical for fixed-width integers and other basic values. Again, an implementation may omit such wrappers at the cost of blame completeness when it detects a type mismatch for an unwrapped value. Because of these challenges, we predict that an implementation of trace wrappers will be blame-sound but blame-incomplete.

10 RELATED WORK

Our work introduces new criteria for analyzing a gradual/migratory typing system, its semantics, and its promises. Hence we first compare our work here to other criteria for gradual typing.

The standard metatheory for gradual typing includes a type soundness theorem and a blame theorem [Tobin-Hochstadt and Felleisen 2006; Wadler 2015; Wadler and Findler 2009]. The latter property says that all blame errors are manifest by a dynamic value flowing into a static context. Siek et al. [2015b] introduce criteria for the meaning of types in a grammar that includes a dynamic type. None of these criteria distinguish a system that fully enforces types (such as Natural) from one that does not (such as Amnesic and Transient).

Three recent pieces of work go beyond the standard metatheory. Chung et al. [2018] compare different gradual typing systems as different translations from one surface language to one core language; they do not use theorems to characterize the difference in type soundness. Greenman and Felleisen [2018] interpret one gradual typing systems as several different semantics for a common surface language; their theorems establish distinct type soundness properties. New et al. [2019] present an axiomatic theory of gradual typing and prove that only the Natural semantics satisfies all the axioms. Other semantics can be understood based on how they fail to satisfy the axioms. The axioms do not specify the blame behavior of gradual typing systems.

Our search for additional criteria started from the work on *complete monitoring* [Dimoulas et al. 2012] and a fundamental insight from the research on higher-order contracts: ***specifications can be wrong***. This point was driven home dramatically when Racket programmers complained about failures of the original contract system [Findler and Felleisen 2002] concerning dependent contracts. On some occasions, these contracts failed to catch a problem; on others, the blame assignment pointed in the wrong direction. Several researchers also noted problems with the original semantics, but their proposed alternatives did not resolve the practical problems either [Blume and McAllester 2006; Greenberg et al. 2012; Hinze et al. 2006].

With complete monitoring, it was possible to pinpoint the faulty assumption behind the original contract work thanks to the crucial technical tool of *ownership semantics* [Dimoulas et al. 2011], which was in turn inspired by a syntactic techique for proving type abstraction [Grossman et al. 2000]. Complete monitoring led to a new implementation of contracts that eliminated the observed problems and helped research teams validate later work on: first-class classes [Takikawa et al. 2012], delimited continuations [Takikawa et al. 2013], and authorization contracts [Moore et al. 2016].

Ownership semantics also served to validate the design of Whip, a contract system that may be partially deployed in a distributed application [Waye et al. 2017]. Although Whip does not satisfy complete monitoring, it does satisfy a weakened blame correctness theorem. Our paper articulates the generalization with blame soundness and blame completeness; Whip is blame-sound relative to a heap-based notion of ownership (section 9.1) and blame-incomplete. On a similar note, we conjecture that the $\overline{\Vdash}_{\mathsf{A}}$ relation for Amnesic answers a question raised by Swords et al. [2018] about how to adapt complete monitoring for "best-effort" contracts; such contracts do not satisfy complete monitoring, but may aim for blame soundness and completeness.

As for the various *semantics* to be considered, there are several we did not analyze here. Castagna and Lanvin [2017] present a semantics that resembles Amnesic without blame. The monotonic semantics for references enforces an increasingly-precise type for every heap location [Siek et al. 2015c]; a monotonic variant of Natural would store functions on a heap and add a monitor directly to a heap value $\mathcal{H}(\mathrm{p})$ when the address p crosses a boundary. Pyret (`pyret.org`) implements a combination of the Natural and Transient semantics without blame; the language eagerly checks fixed-size data and performs first-order checks for recursive types (e.g. lists) and higher-order types.[10] In terms of our framework, Pyret establishes Natural-like boundaries for fixed-size data and permissive ones for everthing else.

Semantics of mixed-typed languages other than the above fall outside the scope of this paper. An *erasure*, or optional, semantics uses types for static analysis and ignores them at runtime. Erasure type soundness, $\mathbf{TS}(\cdot, \mathbf{0})$, is independent of the surface-language type [Bierman et al. 2014; Bonnaire-Sergeant et al. 2016; Chaudhuri et al. 2017; Maidl et al. 2015]. A *concrete* semantics requires a runtime type for every value, and therefore prohibits untyped code from creating new values that flow to typed regions [Muehlboeck and Tate 2017; Rastogi et al. 2015; Richards et al. 2017, 2015; Wrigstad et al. 2010].

11 COMPLETE MONITORING IS NEEDED

Greenman and Felleisen [2018] demonstrated that Typed Racket and Transient Reticulated Python satisfy different type soundness theorems. This paper shows that these languages also differ in how they protect untyped code from faulty type specifications. To explain this difference, we adapt the notion of complete monitoring from contracts to gradual typing.

The adaptation of complete monitoring also provides a tool for explaining the differences in blame between the semantics. While the Natural semantics points to a single boundary between typed and untyped code, Transient delivers a set of boundaries that may point to irrelevant ones and may miss crucial ones, relative to a standard path-based notion of ownership.

The creation of Amnesic illustrates the use of complete monitoring as a design tool. The Amnesic semantics satisfies the same type soundness as Natural and performs the same first-order checks as Transient. Amnesic does not satisfy complete monitoring, but its blame-assignment policy satisfies both soundness and completeness. Although we do not consider Amnesic ready for implementation, its very existence validates the power of complete monitoring as a design guideline.

In sum, we demonstrate that complete monitoring is an effective tool for researchers working on mixed-typed languages. It helps with both the analysis and design of the semantics of typing systems. Adding complete monitoring to the researchers' tool box will greatly improve future explorations of this area.

TECHNICAL REPORT

The technical report is available from the Northeastern University Library Digital Repository Service (DRS) [Greenman et al. 2019a].

ACKNOWLEDGMENTS

We thank Michael M. Vitousek for conversations about Transient and its type soundness; Amal Ahmed, Michael Ballantyne, Stephen Chang, and Max S. New for conversations about complete monitoring; and the anonymous OOPSLA reviewers for their questions, comments, and proofreading. This work was partially supported by NSF grants CCF 1518844 and CCF 1763922.

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

REFERENCES

Esteban Allende, Johan Fabry, and Éric Tanter. 2013. Cast Insertion Strategies for Gradually-Typed Objects. In *DLS*. 27–36.

Henk Barendregt. 1981. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland Publishing Company.

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

Matthias Blume and David A. McAllester. 2006. Sound and Complete Models of Contracts. *JFP* 16, 4-5 (2006), 375–414.

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

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

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.

Sheng Chen and John Peter Campora, III. 2019. Blame Tracking and Type Error Debugging. In *SNAPL*. 2:1–2:14.

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

Matteo Cimini and Jeremy Siek. 2017. Automatically Generating the Dynamic Semantics of Gradually Typed Languages. In *POPL*. 789–803.

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

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

Asger Feldthaus and Anders Møller. 2014. Checking Correctness of TypeScript Interfaces for JavaScript Libraries. In *OOPSLA*. 1–16.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press.

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

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

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2012. Contracts made manifest. *JFP* 22, 3 (2012), 225–274.

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

Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2019a. *Complete Monitors for Gradual Types: Supplementary Material*. Technical Report NU-CCIS-2019-001. Northeastern University.

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

Dan Grossman, Greg Morrisett, and Steve Zdancewic. 2000. Syntactic Type Abstraction. *TOPLAS* 22, 6 (2000), 1037–1080.

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

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

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

Jacob Matthews and Robert Bruce Findler. 2009. Operational Semantics for Multi-Language Programs. *TOPLAS* 31, 3 (2009), 1–44.

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375.

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

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

Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs. *PACMPL* 2, ICFP (2018), 73:1–73:30.

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

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

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.

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

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

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

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

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

Vincent St-Amour and Neil Toronto. 2013. Experience Report: Applying Random Testing to a Base Type Environment. In *ICFP*. 351–356.

Cameron Swords, Amr Sabry, and Sam Tobin-Hochstadt. 2018. An extended account of contract monitoring strategies as patterns of communication. *JFP* 28, e4 (2018), 1–47.

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.

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

Asumu Takikawa, T. Stephen Strickland, and Sam Tobin-Hochstadt. 2013. Constraining Delimited Control with Contracts. In *ESOP*. 229–248.

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

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

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

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

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.

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

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.

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

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

Lucas Waye, Stephen Chong, and Christos Dimoulas. 2017. Whip: Higher-Order Contracts for Modern Services. *PACMPL* 1, ICFP (2017), 36:1–36:28.

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