Monotonic References for Gradual Typing

Jeremy G. Siek Michael M. Vitousek
Matteo Cimini Sam Tobin-Hochstadt
Indiana University Bloomington
jsiek@indiana.edu

Ronald Garcia
University of British Columbia
rxg@cs.ubc.ca

Abstract

Gradual typing enables both static and dynamic typing in the same program, and makes it convenient to migrate code between the two typing disciplines. We have had a satisfactory static semantics for gradual typing for some time but the dynamic semantics has proved much more difficult, raising numerous research challenges. Ongoing efforts to integrate gradual typing into existing functional and object-oriented languages revealed problems regarding space efficiency, run-time overhead, and object identity. While the first problem has been solved, the later two problems remain open. The essence of these problems is best studied in the context of the gradually-typed lambda calculus with mutable references.

In this paper we present a new dynamic semantics called *monotonic references*, which does not require proxies, thereby solving the object identity problem, and is the first to completely eliminate run-time overhead in statically typed code while maintaining the flexibility of gradual typing. With our design, casting a reference may cause a heap cell to become less dynamically typed (but not more). However, retaining type safety is challenging in a semantics such as this that allows strong updates to the heap. Nevertheless, we present a mechanized proof that monotonic references are type safe. We also present a blame tracking strategy for monotonic references and prove the blame theorem.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs – Type structure

General Terms Languages, Theory

Keywords gradual typing, mutable references

1. Introduction

Static and dynamic type systems have well-known strengths and weaknesses. Static type systems provide a machine-checked form of documentation, catch bugs early, and help the compiler generate efficient code. Dynamic type systems provide the flexibility often needed during prototyping and enable powerful features such as reflection. Over the years, many languages have blurred the boundary between static and dynamic typing, such as type hints in Lisp (Steele 1990) and the addition of a dynamic type, named `Dyn`, to otherwise statically typed languages (Abadi et al. 1989). But the seamless integration of static and dynamic typing remained problematic (Thatte 1990; Oliart 1994) until Siek and Taha (2006, 2007) designed a solution named *gradual typing*.

The static semantics of gradual typing has worked well and seen considerable uptake in industry, including Google's Dart language (Bracha 2011), Microsoft's TypeScript (Hejlsberg 2012), and Facebook's variant of PHP (Verlaguet 2013). The run-time checking aspect of gradual typing, which governs the passing of values between static and dynamic regions of a program, has met several challenges and is an ongoing area of research.

The first challenge for the dynamic semantics of gradual typing was defining a notion of type safety that would provide guarantees for the statically-typed part of a program while admitting that the dynamically-typed side can cause run-time cast errors. Tobin-Hochstadt and Felleisen (2006) applied the notion of *blame* from Findler and Felleisen (2002) to characterize these possibilities in their alternative approach to integrating static and dynamic typing. Wadler and Findler (2009) ported this idea to gradual typing, creating the Blame Theorem which pin-points the causes of cast errors and guarantees that "well-typed programs can't be blamed".

Meanwhile, in the design of a gradually-typed JavaScript, Herman and colleagues observed that the function proxies used to implement higher-order casts can take space proportional to execution time (Herman et al. 2007; Herman and Flanagan 2007; Herman et al. 2010). They showed that, in theory, the space efficiency problem could be solved by representing casts with the coercions of Henglein (1994). Siek and Wadler (2010) developed an implementation approach for space efficient casts.

During the design and implementation of Thorn (Bloom et al. 2009), Wrigstad et al. (2010) observed that it is challenging to design a gradually-typed language that does not incur run-time overhead in statically-typed code. To address this problem, they introduce a distinction between *like types* and *concrete types*. Concrete types are the usual types of a statically-typed language and incur zero run-time overhead, but dynamically-typed values cannot flow into concrete types. Like types, on the other hand, provide static checking but incur run-time overhead and may refer to dynamically-typed values. In this paper we propose an alternative solution to the run-time overhead problem that is more expressive, it allows dynamically-typed values to flow into code with (concrete) static types, and that presents a simpler type system to the programmer by only having one kind of type.

Vitousek et al. (2012, 2014) observed another problem during the design and evaluation of gradually-typed Python. Adapting the standard treatment of mutable references (Herman and Flanagan 2007) to objects requires wrapping objects with proxies when they flow through a cast, and proxies cause well-known problems with object identity (Van Cutsem and Miller 2010). In fact, Vitousek et al. (2014) found that adding type annotations to existing Python programs would sometimes cause incorrect program behavior.

In this paper we investigate the essence of these problems in the context of the gradually-typed lambda calculus with mutable references. We present a new dynamic semantics called *monotonic references*, that does not use proxies. Instead, when a reference flows through a cast, the cast may cause the heap cell to become less dynamically typed. That is, the type of a value in the heap monotonically decreases with respect to the less-or-equally-dynamic partial order defined in Figure 1. This relation is also known as naive subtyping (Wadler and Findler 2009), but because it is not used for

$$\begin{array}{lrcl} \text{Base types} & B & ::= & \texttt{Int} \mid \texttt{Bool} \\ \text{Types} & T & ::= & B \mid T \to T \mid T \times T \mid \texttt{Ref}\, T \mid \star \end{array}$$

$$\boxed{T \sqsubseteq T}$$

$$T \sqsubseteq \star \qquad B \sqsubseteq B \qquad \frac{T_1 \sqsubseteq T_2}{\texttt{Ref}\, T_1 \sqsubseteq \texttt{Ref}\, T_2}$$

$$\frac{T_1 \sqsubseteq T_3 \quad T_2 \sqsubseteq T_4}{T_1 \to T_2 \sqsubseteq T_3 \to T_4} \qquad \frac{T_1 \sqsubseteq T_3 \quad T_2 \sqsubseteq T_4}{T_1 \times T_2 \sqsubseteq T_3 \times T_4}$$

Figure 1. Types and the less-or-equally-dynamic relation.

$$\boxed{\mathit{static}\ T}$$

$$\frac{}{\mathit{static}\ B} \qquad \frac{\mathit{static}\ T_1 \quad \mathit{static}\ T_2}{\mathit{static}\ T_1 \to T_2}$$

$$\frac{\mathit{static}\ T_1 \quad \mathit{static}\ T_2}{\mathit{static}\ T_1 \times T_2} \qquad \frac{\mathit{static}\ T}{\mathit{static}\ \texttt{Ref}\, T}$$

Figure 2. Fully-static types.

subsumption and is not naive, we prefer to use a different name. For brevity, we often just say "less dynamic". Figure 1 gives the grammar for types, which include base types (`Int`, `Bool`), function $T \to T$ and pair types $T \times T$, reference types $\texttt{Ref}\, T$, and the unknown type $\star$ (aka. the dynamic type).

Proposition 1 ($\sqsubseteq$ is a partial order)**.**

1. $T \sqsubseteq T$.
2. *If* $T_1 \sqsubseteq T_2$ *and* $T_2 \sqsubseteq T_1$*, then* $T_1 = T_2$.
3. *If* $T_1 \sqsubseteq T_2$ *and* $T_2 \sqsubseteq T_3$*, then* $T_1 \sqsubseteq T_3$.

Monotonic references preserve a global invariant that the type of every references is less-or-equally-dynamic than the type of the value on the heap that it points to. As a result, a fully-static reference always points to a value of the same type, and no casts need be performed while reading or writing through the reference. By "fully static" we mean that there are no occurrences of $\star$ in the type, as defined by the *static* predicate in Figure 2.

Proposition 2 (Fully-static types are the least dynamic)**.**
If $\mathit{static}\ T$ *and* $T' \sqsubseteq T$ *then* $T' = T$.

Thus, there is zero run-time overhead in statically-typed code. On the other hand, reads and writes to references that are not fully-static still require casts.

Swamy et al. (2014) independently invented the use of a monotonic heap for gradual typing, but for a different reason and with respect to a different ordering. Their motivation was to provide security guarantees, not remove overhead, and their monotonicity is with respect to subtyping instead of the less-dynamic relation.

While monotonic references solve several problems, they do come with a cost. The monotonic semantics is more restrictive than the standard one, sometimes triggering a cast error when the standard semantics would not. However, in concurrent code, the standard semantics can exhibit non-deterministic behavior, triggering a cast error only with certain interleavings of thread execution, whereas the monotonic semantics will always trigger a cast error in such situations (see Section 2.3). On the whole, we believe monotonic references are the right design choice in scenarios where high-performance is a priority and where programmers intend to eventually convert their programs to be statically typed. Monotonic references are no more restrictive than those of statically-typed languages such as ML.

The dynamic semantics of monotonic references is particularly subtle because references may point to values that themselves contain references, and furthermore, the dynamic points-to graph may be cyclic. Thus, applying a cast to a reference requires what amounts to a fixed-point computation on a sub-region of the heap. Also, because we change heap values to have different types, that is, we perform *strong updates*, it is non-trivial to prove type safety. Nevertheless, we present a mechanized proof of type safety.

In gradually-typed languages with higher-order features such as first-class functions and objects, blame tracking plays an important role in providing meaningful error messages when casts fail and it enables fine-grained guarantees, via the blame theorem, regarding which regions of the code are statically type safe. In this paper we present blame tracking for monotonic references and prove the blame theorem. Designing a blame tracking strategy for monotonic references was a multi-year effort involving the exploration of several alternatives. The key to our design is to use the labeled types of Siek and Wadler (2010) as run-time type information (RTTI), together with three new operations on labeled types: a bidirectional cast operator that captures the dual read/write nature of mutable references, a merge operator that models how casts on separate aliases to the same heap cell interact over time, and an operator that casts heap cells from one labeled type to another via a coercion.

To summarize, this paper presents a new dynamic semantics for gradually-typed mutable references that finally delivers efficiency for the statically-typed parts of a program, maintains type soundness, provides blame tracking, and relieves the problems with object identity. This result may improve the design and implementation of gradual typing for functional languages such as Racket and Clojure as well as object-oriented languages such as TypeScript, Python, and PHP. More concretely, this paper makes the following three technical contributions:

1. We define the dynamic semantics for monotonic references, the first system to simultaneously solve the object-identity and run-time overhead problems while maintaining the expressiveness of gradual typing (Sections 3 and 5).
2. We mechanize a proof of type safety in Isabelle (Section 4).
3. We augment monotonic references with blame tracking and prove the blame theorem (Section 6). The supplemental material includes an interpreter with this blame tracking strategy.

We review the gradually-typed lambda calculus with references in Section 2 and discuss the problems of object identity and run-time overhead. We address an implementation concern regarding strong updates in Section 7 and we discuss related work in Section 8. The paper concludes in Section 9.

2. Background and Problem Statement

Figure 3 reviews the syntax and static semantics of the gradually-typed lambda calculus with references. The primary difference between gradual typing and simple typing is that uses of type equality are replaced by *consistency*, also defined in Figure 3. The consistency relation enables implicit casts to and from $\star$. (In contrast, an object-oriented language would only allow implicit casts to $\star$.) The consistency relation is a congruence, even for reference types (Herman et al. 2010). (Siek and Taha (2006) used an invariant rule for references.) This treatment of references enables the passing of references between more and less statically typed regions code, but is also the source of the difficulties that we solve in this paper. The relations $\mathit{fun}, \mathit{pair}$, and ref, defined in Figure 3, implement pattern matching on types, which enables a more concise presentation of the typing rules. We include labels ℓ in the syntax to represent source code locations that would be added during parsing. For an introduction to gradual type systems and examples, we refer

Syntax

$$
\begin{array}{llll}
\text{Labels} & \ell & & \\
\text{Operators} & op & ::= & \texttt{plus} \mid \texttt{minus} \mid \texttt{is} \mid \cdots \\
\text{Expressions} & e & ::= & k \mid op^{\ell}(\vec{e}) \mid x \mid \lambda x{:}T.\, e \mid (e\ e)^{\ell} \mid e\ \texttt{as}^{\ell}\ T \mid \\
 & & & (e, e) \mid \texttt{fst}^{\ell}\, e \mid \texttt{snd}^{\ell}\, e \mid \\
 & & & \texttt{ref}\ e \mid\ !^{\ell} e \mid e\ \texttt{:=}^{\ell}\ e \\
 & & & \lambda x.\, e \equiv \lambda x{:}\star.\, e
\end{array}
$$

Consistency $\boxed{T \sim T}$

$$\frac{}{\star \sim T} \qquad \frac{}{T \sim \star} \qquad \frac{}{B \sim B} \qquad \frac{T_1 \sim T_2}{\texttt{Ref}\ T_1 \sim \texttt{Ref}\ T_2}$$

$$\frac{T_1 \sim T_3 \quad T_2 \sim T_4}{T_1 \to T_2 \sim T_3 \to T_4} \qquad \frac{T_1 \sim T_3 \quad T_2 \sim T_4}{T_1 \times T_2 \sim T_3 \times T_4}$$

Expression typing $\boxed{\Gamma \vdash e : T}$

$$\frac{k : B}{\Gamma \vdash k : B} \qquad \frac{\Gamma \vdash \vec{e} : \vec{T} \quad op : \vec{B} \to B \quad \vec{T} \sim \vec{B}}{\Gamma \vdash op^{\ell}(\vec{e}) : B}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\Gamma(x \mapsto T_1) \vdash e : T_2}{\Gamma \vdash \lambda x{:}T_1.\, e : T_1 \to T_2}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad \mathit{fun}(T_1, T_{11}, T_{12}) \quad T_2 \sim T_{11}}{\Gamma \vdash (e_1\ e_2)^{\ell} : T_{12}}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : T_1 \times T_2} \qquad \frac{\Gamma \vdash e : T \quad \mathit{pair}(T, T_1, T_2)}{\Gamma \vdash \texttt{fst}^{\ell}\, e : T_1} \qquad \frac{\Gamma \vdash e : T \quad \mathit{pair}(T, T_1, T_2)}{\Gamma \vdash \texttt{snd}^{\ell}\, e : T_2}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \texttt{ref}\ e : \texttt{Ref}\ T} \qquad \frac{\Gamma \vdash e : T \quad \mathit{ref}(T, T')}{\Gamma \vdash\ !^{\ell} e : T'}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad \mathit{ref}(T_1, T_1') \quad T_2 \sim T_1'}{\Gamma \vdash e_1\ \texttt{:=}^{\ell}\ e_2 : T_1} \qquad \frac{\Gamma \vdash e : T_1 \quad T_1 \sim T_2}{\Gamma \vdash e\ \texttt{as}^{\ell}\ T_2 : T_2}$$

Type matching

$$\frac{}{\mathit{fun}(T_{11} \to T_{12}, T_{11}, T_{12})} \qquad \frac{}{\mathit{fun}(\star, \star, \star)}$$

$$\frac{}{\mathit{pair}(T_{11} \times T_{12}, T_{11}, T_{12})} \qquad \frac{}{\mathit{pair}(\star, \star, \star)}$$

$$\frac{}{\mathit{ref}(\texttt{Ref}\ T, T)} \qquad \frac{}{\mathit{ref}(\star, \star)}$$

Figure 3. Gradually-typed λ calculus with mutable references

the reader to prior literature (Siek and Taha 2006, 2007; Tobin-Hochstadt and Felleisen 2006, 2008; Herman et al. 2010).

The dynamic semantics of the gradually-typed lambda calculus is defined by a type-directed translation to the coercion calculus (Henglein 1994). Each use of consistency between types T_1 and T_2 in the type system, and each use of one of the auxiliary relations, becomes an explicit cast from T_1 to T_2. The coercion calculus expresses casts in terms of combinators that say how to cast from one type to another. Figure 4 gives the compilation of casts into coercions, written $(T \Rightarrow^{\ell} T) = c$. The compilation of gradually-typed terms into the coercion calculus is otherwise straightforward, so we give just one rule as an example, for function application:

$$\frac{\begin{array}{cc}\Gamma \vdash e_1 \rightsquigarrow e_1' : T_1 & \Gamma \vdash e_2 \rightsquigarrow e_2' : T_2 \\ \mathit{fun}(T_1, T_{11}, T_{12}) & T_2 \sim T_{11} \\ (T_1 \Rightarrow^{\ell} T_{11} \to T_{12}) = c_1 & (T_2 \Rightarrow^{\ell} T_{11}) = c_2\end{array}}{\Gamma \vdash (e_1\ e_2)^{\ell} \rightsquigarrow e_1'\langle c_1 \rangle\ e_2'\langle c_2 \rangle : T_{12}}$$

$\boxed{(T \Rightarrow^{\ell} T) = c}$

$$
\begin{array}{rlrl}
(B \Rightarrow^{\ell} B) &= \iota & (I \Rightarrow^{\ell} \star) &= I! \\
(\star \Rightarrow^{\ell} \star) &= \iota & (\star \Rightarrow^{\ell} I) &= I?^{\ell} \\
\multicolumn{4}{c}{(T_1 \to T_2) \Rightarrow^{\ell} (T_1' \to T_2') = (T_1' \Rightarrow^{\ell} T_1) \to (T_2 \Rightarrow^{\ell} T_2')} \\
\multicolumn{4}{c}{(T_1 \times T_2) \Rightarrow^{\ell} (T_1' \times T_2') = (T_1 \Rightarrow^{\ell} T_1') \times (T_2 \Rightarrow^{\ell} T_2')} \\
\multicolumn{4}{c}{\texttt{Ref}\ T \Rightarrow^{\ell} \texttt{Ref}\ T' = \texttt{Ref}\ (T \Rightarrow^{\ell} T')\ (T' \Rightarrow^{\ell} T)}
\end{array}
$$

Figure 4. Compile casts to coercions

Figure 5 defines the coercion calculus following a recently simplified presentation (Anonymous 2014) but implementing the D blame tracking semantics of Siek et al. (2009). We highlight the parts of the definition related to references, as they are of particular interest here. We review the coercion calculus in the context of discussing the object identity and run-time overhead problems in the following subsections. For an introduction to the coercion calculus, we refer to Henglein (1994) and Siek et al. (2009).

2.1 The problem with object identity

Consider the following $move$ function that transfers an integer from one heap cell to another and zeroes out the source cell. The function starts by checking whether reference x and y are aliased using the `is` operator, in which case it does nothing. After the definition of $move$, we allocate a single cell, storing the address in reference r, and call $move$ with two occurrences of r.

$$
\begin{array}{l}
\texttt{let}\ move = \lambda x{:}\texttt{Ref Int}.\, \lambda y{:}\texttt{Ref Int}. \\
\quad \texttt{if}\ \texttt{is}^{\ell_4}(x, y)\ \texttt{then}\ () \\
\quad\ \texttt{else}\ x\ \texttt{:=}^{\ell_5}\ !y; y\ \texttt{:=}^{\ell_6}\ 0; ()\ \texttt{in} \\
\texttt{let}\ r = \texttt{ref}\ 42\ \texttt{in} \\
((move\ r)^{\ell_2}\ r)^{\ell_3}; !r
\end{array}
$$

The result of the above program is 42. Next suppose that we change the fourth line so that the reference r is passed through some dynamically-typed code, which we model by casting it to `Ref` $\star$.

$$\texttt{let}\ r = (\texttt{ref}\ 42)\ \texttt{as}^{\ell_1}\ \texttt{Ref}\ \star\ \texttt{in}$$

The cast from `Ref Int` to `Ref` $\star$ compiles to `Ref Int! Int`$?^{\ell_1}$. The first coercion `Int!`, an injection, is applied when reading from the reference, casting from `Int` to $\star$, and the second coercion `Int`$?^{\ell_1}$, a projection, is applied when writing to the reference, casting from $\star$ to `Int`. Applying the reference coercion to the address a produced by `ref` 42 produces $v_1 = a\langle \texttt{Ref Int! Int}?^{\ell_1} \rangle$, that is, a proxied reference. There are also implicit casts in the call to $move$, from `Ref` $\star$ to `Ref Int`. Each parameter is wrapped in a cast, producing $v_2 = v_1\langle \texttt{Ref Int}?^{\ell_2}\ \texttt{Int!} \rangle$ and $v_3 = v_1\langle \texttt{Ref Int}?^{\ell_3}\ \texttt{Int!} \rangle$. Proceeding to the body of $move$, we come to the interesting question: what should $\texttt{is}(v_2, v_3)$ return?

One option is for the `is` operator to compare the underlying addresses. However, it is desirable to only have `is` return true when the two references are behaviorally equivalent. But differently-wrapped addresses may have different behavior, for example, one may trigger an error when the other does not. Another option is for `is` to compare the addresses and the coercions, that is, adapt the *membrane* solution proposed by Van Cutsem and Miller (2010). However, this approach would cause $\texttt{is}(v_2, v_3)$ to return false, changing the result of the above program from 42 to 0. The problem with respect to gradual typing is that membranes preserve identity within a single membrane but not across different membranes, which in this setting are just different casts. Alternatively, if proxies had addresses, one could compare the proxy addresses. But that also changes the result to 0. In short, these approaches break a fun-

Syntax

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & k \mid op(\vec{e}) \mid x \mid \lambda x.\, e \mid e\ e \mid \\
 & & & (e, e) \mid \texttt{fst}\, e \mid \texttt{snd}\, e \mid \\
 & & & \texttt{ref}\, e \mid \texttt{!}e \mid e\ \texttt{:=}\ e \mid e\langle c \rangle \mid \texttt{blame}\, \ell \\
\text{Injectibles} & I & ::= & B \mid T \to T \mid T \times T \mid \texttt{Ref}\, T \\
\text{Coercions} & c & ::= & \iota \mid I?^{\ell} \mid I! \mid c \to c \mid c \times c \mid c\,;c \mid \texttt{Ref}\, c\, c \\
\text{Values} & v & ::= & k \mid \lambda x.\, e \mid (v, v) \mid v\langle I! \rangle \mid a \mid v\langle \texttt{Ref}\, c\, c \rangle \\
\text{Heap} & \mu & ::= & \emptyset \mid \mu(a \mapsto v) \\
\text{Heap Typing} & \Sigma & ::= & \emptyset \mid \Sigma(a \mapsto T) \\
\text{Frames} & F & ::= & op(\vec{v}, \square, \vec{e}) \mid \square\ e \mid v\ \square \mid \\
 & & & (\square, e) \mid (v, \square) \mid \texttt{fst}\, \square \mid \texttt{snd}\, \square \mid \\
 & & & \texttt{ref}\, \square \mid \texttt{!}\square \mid \square\ \texttt{:=}\ e \mid v\ \texttt{:=}\ \square \mid \square\langle c \rangle
\end{array}
$$

Coercion typing $\boxed{c : T \Rightarrow T}$

$$\frac{}{\iota : T \Rightarrow T} \qquad \frac{c_1 : T_3 \Rightarrow T_1 \quad c_2 : T_2 \Rightarrow T_4}{c_1 \to c_2 : (T_1 \to T_2) \Rightarrow (T_3 \to T_4)}$$

$$\frac{}{I?^{\ell} : \star \Rightarrow I} \qquad \frac{c_1 : T_1 \Rightarrow T_3 \quad c_2 : T_2 \Rightarrow T_4}{c_1 \times c_2 : (T_1 \times T_2) \Rightarrow (T_3 \times T_4)}$$

$$\frac{}{I! : I \Rightarrow \star} \qquad \frac{c_1 : T_1 \Rightarrow T_2 \quad c_2 : T_2 \Rightarrow T_3}{c_1\,;c_2 : T_1 \Rightarrow T_3}$$

$$\frac{c_1 : T_1 \Rightarrow T_2 \quad c_2 : T_1 \Rightarrow T_2}{\texttt{Ref}\, c_1\, c_2 : \texttt{Ref}\, T_1 \Rightarrow \texttt{Ref}\, T_2}$$

Expression typing $\boxed{\Gamma; \Sigma \vdash e : T}$

$$\ldots \qquad \frac{\Sigma(a) = T}{\Gamma; \Sigma \vdash a : T} \qquad \frac{\Gamma; \Sigma \vdash e : T_1 \quad c : T_1 \Rightarrow T_2}{\Gamma; \Sigma \vdash e\langle c \rangle : T_2}$$

Reduction rules for functions, primitives, and pairs $\boxed{e \longrightarrow e}$

$$
\begin{array}{llll}
(\lambda x.\, e)\ v & \longrightarrow [x := v]e & \texttt{fst}\,(v_1, v_2) & \longrightarrow v_1 \\
op(\vec{k}) & \longrightarrow \delta(op, \vec{k}) & \texttt{snd}\,(v_1, v_2) & \longrightarrow v_2
\end{array}
$$

Cast reduction rules $\boxed{e \longrightarrow_c e}$

$$
\begin{array}{rl}
v\langle \iota \rangle & \longrightarrow_c v \\
v\langle I_1! \rangle \langle I_2?^{\ell} \rangle & \longrightarrow_c v\langle I_1 \Rightarrow^{\ell} I_2 \rangle \quad \text{if } I_1 \sim I_2 \\
v\langle I_1! \rangle \langle I_2?^{\ell} \rangle & \longrightarrow_c \texttt{blame}\, \ell \quad \text{if } I_1 \not\sim I_2 \\
v\langle c_1 \to c_2 \rangle & \longrightarrow_c \lambda x.\, v\,(x\langle c_1 \rangle)\langle c_2 \rangle \\
(v_1, v_2)\langle c_1 \times c_2 \rangle & \longrightarrow_c (v_1\langle c_1 \rangle, v_2\langle c_2 \rangle) \\
v\langle c_1\,;c_2 \rangle & \longrightarrow_c v\langle c_1 \rangle \langle c_2 \rangle
\end{array}
$$

Reference reduction rules $\boxed{e, \mu \longrightarrow_r e, \mu}$

$$
\begin{array}{rll}
\texttt{ref}\, v, \mu & \longrightarrow_r a, \mu(a \mapsto v) & \text{if } a \notin dom(\mu) \quad (1) \\
\texttt{!}a, \mu & \longrightarrow_r \mu(a), \mu & (\textsc{Deref}) \\
\texttt{!}(v\langle \texttt{Ref}\, c_1\, c_2 \rangle) & \longrightarrow_r (\texttt{!}v)\langle c_1 \rangle & (\textsc{DerefCast}) \\
a\ \texttt{:=}\ v, \mu & \longrightarrow_r a, \mu(a \mapsto v) & (\textsc{Update}) \\
v_1\langle \texttt{Ref}\, c_1\, c_2 \rangle\ \texttt{:=}\ v_2 & \longrightarrow_r v_1\ \texttt{:=}\ v_2\langle c_2 \rangle & (\textsc{UpdateCast})
\end{array}
$$

State reduction rules

$$\frac{e \longrightarrow e'}{e, \mu \longrightarrow e', \mu} \qquad \frac{e \longrightarrow_c e'}{e, \mu \longrightarrow e', \mu} \qquad \frac{e, \mu \longrightarrow_r e', \mu'}{e, \mu \longrightarrow e', \mu'}$$

$$\frac{e, \mu \longrightarrow e', \mu'}{F[e], \mu \longrightarrow F[e'], \mu'} \qquad \frac{}{F[\texttt{blame}\, \ell], \mu \longrightarrow \texttt{blame}\, \ell, \mu}$$

Figure 5. Coercion calculus with mutable references

damental property of casts: that adding casts to a program should not change the behavior other than to induce more cast errors. In this paper we investigate a design that does not use proxies.

2.2 Run-time overhead in fully-static code

Consider the following fully-static function f that dereferences its parameter x.

$$
\begin{array}{l}
\texttt{let}\ f = \lambda x{:}\mathsf{Ref\ Int}.\ \texttt{!}^{\ell_3} x\ \texttt{in} \\
f(\texttt{ref}\ 4); \\
f(\texttt{ref}\ (\texttt{true}\ \texttt{as}^{\ell_1}\ \star)\ \texttt{as}^{\ell_2}\ \texttt{Ref Int})
\end{array}
$$

In the first call to f, a normal reference to an integer flows into the dereference of x whereas in the second call, a proxied reference flows into the dereference of x (under the standard semantics). The code generated for the dereference in the body of f needs to be general enough to handle both kinds of references. The code must inspect the value and dispatch, thereby incurring run-time overhead. The overhead can also be seen in the dynamic semantics (Figure 5), where there are two reduction rules for dereferencing: (DEREF) and (DEREFCAST), and two reduction rules for updating references: (UPDATE) and (UPDATECAST). Another way to look at this problem is that there are two canonical forms of type `Ref Int`, a plain address a and also a value wrapped in a reference coercion, $v\langle \texttt{Ref}\, c_1\ c_2 \rangle$. To get rid of the overhead we need a design with only a single canonical form for values of reference type.

The run-time overhead for references affects every read and write to the heap and is particularly detrimental in tight loops over arrays. When adding support for contracts to mutable data structures in Racket, Strickland et al. (2012, Figure 9) measured this overhead at approximately 25% for fully-typed code on a bubble-sort microbenchmark.

2.3 Non-determinism in multi-threaded code

The standard semantics for mutable references originally proposed by Herman et al. (2010) produces an error only if type inconsistency is witnessed by some read or write to the reference, so in a non-deterministic multi-threaded program, whether a check will fail at runtime is difficult to predict.

The contract system in Racket currently implements the standard semantics (Flatt and PLT 2014). For example, the following Racket program sometimes fails blaming `b1`, sometimes fails blaming `b2`, and sometimes succeeds, as explained below.

```
#lang racket
(define b (box #f))
(define/contract b1 (box/c integer?) b)
(define/contract b2 (box/c string?)  b)

(thread (lambda ()
          (for ([i 2])
            (set-box! b1 5)
            (sleep 0.000000001)
            (add1 (unbox b1)))))
(thread (lambda ()
          (for ([i 2])
            (set-box! b2 "hello")
            (sleep 0.000000001)
            (string-append "world" (unbox b2)))))
```

The program creates a single reference cell `b`, and accesses it through two distinct proxies, `b1` and `b2`, each with its own dynamic check. When the two threads do not interleave, the program succeeds, but if the second thread changes `b2` to contain a string between the `set-box!` and `unbox` calls for `b1`, the system halts, blaming one of the parties.

In contrast, if `box/c` implemented monotonic references, then an error would *deterministically* occur when `define/contract` is used for the second time.

3. Monotonic References Without Blame

Figure 6 defines the syntax and semantics of our new coercion calculus with monotonic references, but without blame. Figure 8 defines the compilation of casts to monotonic coercions, also without blame. The addition of blame adds considerable complexity, so we postpone its treatment to Section 5.

With monotonic references, there is only one kind of value at reference type: normal addresses. When a cast is applied to a reference, instead of wrapping the reference with a cast, we cast the underlying value on the heap. To make sure that the new type of the value is consistent with all the outstanding references, we require that a cast only make the type of the value less dynamic (Figure 1). Otherwise the cast results in a runtime error. Thus, we maintain the heap invariant that the type of every reference in the program is less or equally dynamic than the type of the value on the heap that it points to, as captured in the typing rule (WTREF).

The syntax of the monotonic calculus differs from the standard calculus with references in that there are two kinds of dereference and update expressions. We reserve the forms $!e$ and e_1 `:=` e_2 for situations in which the reference type is fully static (Figure 2 and expression typing in Figure 6). In these situations we know that the value in the heap has the same type as the reference thanks to Proposition 2 and our heap invariant. Thus, if a reference has a fully static type, such as `Ref Int`, the corresponding value on the heap must be an actual integer (and not an injection to $\star$), so we need only one reduction rule for dereferencing a fully-static reference (DEREFM), and one rule for updating a fully-static reference (UPDM).

For expressions of reference type that are not fully-static, we introduce the syntactic forms $!e@T$ and e_1 `:=` $e_2@T$ for dereference and update, respectively. The type annotation T records the compile-time type of e, that is, e has type `Ref` T. For example, T could be $\star$, $\star\times\star$, or $\star\times$`Int`. Because the value on the heap might be less dynamic than T, a cast is needed to mediate between T and the run-time type of the heap cell. The reduction rule (DYNDEREFM) casts from the addresses' run-time type, which we store next to the heap cell, to the compile-time type T. We write $\mu(a)_{\mathsf{rtti}}$ for the run-time type information for reference a and we write $\mu(a)_{\mathsf{val}}$ for the value in the heap cell. The reduction rule (DYNUPDM) casts the to-be-written value v from T to the address's run-time type, so the new contents of the cell is $cv = v\langle T \Rightarrow \mu(a)_{\mathsf{rtti}}\rangle$. This cv is not a value yet, so storing it in the heap is unusual. In earlier versions of the semantics we tried to reduce cv to a value before storing it in the heap, but there are complications that force this design, which we discuss later in this section . To summarize our treatment of dereference and update, we present efficient semantics for the fully-static dereference and update but have slightly increased the overhead for dynamic dereferences and updates. This is a price we are willing to pay to have dynamic typing "pay its own way".

The crux of the monotonic semantics is in the reduction rules that apply a reference coercion to an address: (CASTREF1), (CASTREF2), and (CASTREF3). In (CASTREF1) we have an address that maps to cv of type T_1 and we cast cv so that it is less or equally dynamic than both the target type T_2 and all of the existing references to the cell. To accomplish this, we take the greatest lower bound $T_3 = T_1 \sqcap T_2$ (Figure 7) to be the new type of the cell, so the new contents is $cv' = cv\langle T_1 \Rightarrow T_3\rangle$. There are two side conditions on (CASTREF1): $T_1 \sqcap T_2$ must be defined and $T_3 \neq T_1$. If $T_1 \sqcap T_2$ is undefined, or equivalently, if $T_1 \not\sim T_2$, we instead signal an error, as handled by (CASTREF3). If $T_3 = T_1$, then there is no need to cast cv, which is handled by (CASTREF2).

Expressions	e	::=	.. | `ref`$_T\ e$ | $!e@T$ | e `:=` $e@T$ | `error`
Coercions	c	::=	ι | $I?$ | $I!$ | $c{\to}c$ | $c \times c$ | $c\,;c$ | `Ref` T
Values	v	::=	k | $\lambda x.\, e$ | (v, v) | $v\langle I!\rangle$ | a
Casted Values	cv	::=	v | $v\langle c\rangle$ | (cv, cv)
Heap	μ	::=	$\emptyset$ | $\mu(a \mapsto v : T)$
Evolving Heap	ν	::=	$\emptyset$ | $\nu(a \mapsto cv : T)$
Frames	F	::=	.. | $!\Box@T$ | $\Box$ `:=` $e@T$ | v `:=` $\Box@T$

Expression typing $\boxed{\Gamma;\Sigma \vdash e : T}$

$$\frac{\Gamma;\Sigma \vdash e : \texttt{Ref}\ T \quad \mathit{static}\ T}{\Gamma;\Sigma \vdash\ !e : T} \qquad \frac{\Gamma;\Sigma \vdash e_1 : \texttt{Ref}\ T \quad \Gamma;\Sigma \vdash e_2 : T \quad \mathit{static}\ T}{\Gamma;\Sigma \vdash e_1\ \texttt{:=}\ e_2 : \texttt{Ref}\ T}$$

$$\frac{\Gamma;\Sigma \vdash e : \texttt{Ref}\ T}{\Gamma;\Sigma \vdash\ !e@T : T} \qquad \frac{\Gamma;\Sigma \vdash e_1 : \texttt{Ref}\ T \quad \Gamma;\Sigma \vdash e_2 : T}{\Gamma;\Sigma \vdash e_1\ \texttt{:=}\ e_2@T : \texttt{Ref}\ T}$$

$$\cdots \quad \frac{\Sigma(a) \sqsubseteq T_2}{\Gamma;\Sigma \vdash a : T_2} \qquad \text{(WTREF)}$$

Cast reduction rules $\boxed{e,\nu \longrightarrow_{cr} e,\nu}$

$$\frac{e \longrightarrow_c e'}{e,\nu \longrightarrow_{cr} e',\nu} \qquad \text{(PURECAST)}$$

$$\frac{\nu(a) = cv : T_1 \quad T_3 = T_1 \sqcap T_2 \quad T_3 \neq T_1 \quad cv' = cv\langle T_1 \Rightarrow T_3\rangle}{a\langle \texttt{Ref}\ T_2\rangle,\nu \longrightarrow_{cr} a,\nu(a \mapsto cv' : T_3)} \qquad \text{(CASTREF1)}$$

$$\frac{\nu(a) = cv : T_1 \quad T_1 = T_1 \sqcap T_2}{a\langle \texttt{Ref}\ T_2\rangle,\nu \longrightarrow_{cr} a,\nu} \qquad \text{(CASTREF2)}$$

$$\frac{\nu(a) = cv : T_1 \quad T_1 \not\sim T_2}{a\langle \texttt{Ref}\ T_2\rangle,\nu \longrightarrow_{cr} \texttt{error},\nu} \qquad \text{(CASTREF3)}$$

Program reduction rules $\boxed{e,\mu \longrightarrow_e e,\nu}$

$$e,\mu \longrightarrow_e e',\mu \quad \text{if } e \longrightarrow e'$$

$$\texttt{ref}_T\ v,\mu \longrightarrow_e a,\mu(a \mapsto v : T) \quad \text{if } a \notin \mathit{dom}(\mu)$$

$$!a,\mu \longrightarrow_e \mu(a)_{\mathsf{val}},\mu \qquad \text{(DEREFM)}$$

$$!a@T,\mu \longrightarrow_e \mu(a)_{\mathsf{val}}\langle \mu(a)_{\mathsf{rtti}} \Rightarrow T\rangle,\mu \qquad \text{(DYNDEREFM)}$$

$$a\ \texttt{:=}\ v,\mu \longrightarrow_e a,\mu(a \mapsto v : \mu(a)_{\mathsf{rtti}}) \qquad \text{(UPDM)}$$

$$a\ \texttt{:=}\ v@T,\mu \longrightarrow_e a,\mu(a \mapsto cv : \mu(a)_{\mathsf{rtti}}) \qquad \text{(DYNUPDM)}$$

where $cv = v\langle T \Rightarrow \mu(a)_{\mathsf{rtti}}\rangle$

For $X \in \{cr, e\}$:

$$\frac{e,\nu \longrightarrow_X e',\nu'}{F[e],\nu \longrightarrow_X F[e'],\nu'} \qquad \frac{}{F[\texttt{error}],\nu \longrightarrow_X \texttt{error},\nu}$$

State reduction rules $\boxed{e,\nu \longrightarrow e,\nu}$

$$\frac{e,\mu \longrightarrow_X e',\nu \quad X \in \{cr, e\}}{e,\mu \longrightarrow e',\nu}$$

$$\frac{\nu(a) = cv : T \quad cv,\nu \longrightarrow_{cr} cv',\nu' \quad \nu'(a)_{\mathsf{rtti}} = T}{e,\nu \longrightarrow e,\nu'(a \mapsto cv' : T)} \qquad \text{(HCAST)}$$

$$\frac{\nu(a) = cv : T \quad cv,\nu \longrightarrow_{cr} cv',\nu' \quad \nu'(a)_{\mathsf{rtti}} \neq T}{e,\nu \longrightarrow e,\nu'} \qquad \text{(HDROP)}$$

$$\frac{\nu(a) = cv : T \quad cv,\nu \longrightarrow_{cr} \texttt{error},\nu'}{e,\nu \longrightarrow \texttt{error},\nu'} \qquad \text{(HERR)}$$

Figure 6. Monotonic references without blame

$$\boxed{T \sqcap T = T}$$

$$\begin{aligned}
\star \sqcap T &= T \\
T \sqcap \star &= T \\
B \sqcap B &= B \\
(T_1 \times T_2) \sqcap (T_3 \times T_4) &= (T_1 \sqcap T_3) \times (T_2 \sqcap T_4) \\
(T_1 \to T_2) \sqcap (T_3 \to T_4) &= (T_1 \sqcap T_3) \to (T_2 \sqcap T_4)
\end{aligned}$$

Figure 7. The meet function (greatest lower bound)

The rest of the coercion reduction rules are captured by the rule (PURECAST), so they are the same as in the standard semantics (Figure 5), though here we ignore blame, i.e., replace $\texttt{blame}\,\ell$ with $\texttt{error}$, $I_2?^{\ell}$ with $I_2?$, and $I_1 \Rightarrow^{\ell} I_2$ with $I_1 \Rightarrow I_2$.

The meet function defined in Figure 7 indeed computes the greatest lower bound with respect to the less-dynamic relation.

Proposition 3 (Meet computes the greatest lower bound)**.**

1. $(T_1 \sqcap T_2) \sqsubseteq T_1$ *and* $(T_1 \sqcap T_2) \sqsubseteq T_2$.
2. *If* $T \sqsubseteq T_1$ *and* $T \sqsubseteq T_2$*, then* $T \sqsubseteq T_1 \sqcap T_2$.

To motivate our organization of the heap, we present two examples that demonstrate why we store run-time type information in the heap and why we store casted values and not just values on the heap.

Cycles and termination The first complication is that there can be cycles in the heap and we need to make sure that when we apply a cast to an address in a cycle, the cast terminates. Consider the following example in which we create a pair whose second element is a reference back to itself.

$$\begin{array}{l}
\texttt{let}\; r_1 = \texttt{ref}\,(42, 0\;\texttt{as}\;\star)\;\texttt{in} \\
r_1\;\texttt{:=}\,(42, r_1\;\texttt{as}\;\star); \\
\texttt{let}\; r_2 = r_1\;\texttt{as}\;\texttt{Ref}\,(\texttt{Int} \times \texttt{Ref}\,\star)\;\texttt{in} \\
\texttt{fst}\;!r_2
\end{array}$$

Once the cycle is established, we cast r_1 from type $\texttt{Ref}\,(\texttt{Int} \times \star)$ to $\texttt{Ref}\,(\texttt{Int} \times \texttt{Ref}\,\star)$. The presence of the nested $\texttt{Ref}\,\star$ in the target type means that the cast on r_1 will trigger another cast on r_1. The correct result of this program is 42 but a naive dynamic semantics would diverge. Our semantics avoids divergence by checking whether the new run-time type is equal to the old run-time type; in such cases the heap cell is left unchanged (see rule (CASTREF2)).

Casted values in the heap Consider the following example in which we create a triple of type $\star \times \star \times \star$ whose third element is a reference back to itself.

$$\begin{array}{l}
\texttt{let}\; r_0 = \texttt{ref}\,(42\;\texttt{as}\;\star, 7\;\texttt{as}\;\star, 0\;\texttt{as}\;\star)\;\texttt{in} \\
r_0\;\texttt{:=}\,(42\;\texttt{as}\;\star, 7\;\texttt{as}\;\star, r_0\;\texttt{as}\;\star); \\
\texttt{let}\; r_1 = r_0\;\texttt{as}\;\texttt{Ref}\,(\texttt{Int} \times \star \times \texttt{Ref}\,(\texttt{Int} \times \texttt{Int} \times \star))\;\texttt{in} \\
\texttt{fst}\,(\texttt{fst}\;!r_1)
\end{array}$$

Suppose a_0 is the address created in the allocation on the first line. On line three we cast a_0 in such a way that we trigger two casts on a_0. Consider the action of these casts on just the first two elements of the triple, we have:

$$\star \times \star \Rightarrow \texttt{Int} \times \star \Rightarrow \texttt{Int} \times \texttt{Int}$$

The second cast occurs while the first cast is still in progress. Now, suppose we delayed updating the heap cell until we finished reducing to a value. At the moment when we apply the second cast, we would still have the original value, of type $\star \times \star$, in the heap. This is problematic because our next step would be to apply a cast from $\texttt{Int} \times \star \Rightarrow \texttt{Int} \times \texttt{Int}$ to this value, but the value's type and the source type of the cast don't match! In fact, in this example

$$\boxed{(T \Rightarrow T) = c}$$

$$\begin{aligned}
(B \Rightarrow B) &= \iota \\
(\star \Rightarrow \star) &= \iota \\
(I \Rightarrow \star) &= I! \\
(\star \Rightarrow I) &= I? \\
(T_1 \to T_2) \Rightarrow (T_1' \to T_2') &= (T_1' \Rightarrow T_1) \to (T_2 \Rightarrow T_2') \\
(T_1 \times T_2) \Rightarrow (T_1' \times T_2') &= (T_1 \Rightarrow T_1') \times (T_2 \Rightarrow T_2') \\
\texttt{Ref}\,T \Rightarrow \texttt{Ref}\,T' &= \texttt{Ref}\,T'
\end{aligned}$$

Figure 8. Compile casts to monotonic coercions (without blame)

the result would be incorrect; we would get $42\langle\texttt{Int!}\rangle$ as a result instead of 42.

There are several possible solutions to this problem, and they all require storing more information on the heap or as a separate map. Here we take the most straightforward approach of immediately updating the heap with casted values, that is, with values that are in the process of being cast.

We walk through the execution of the above example, explaining our rules for reducing casted values in the heap and showing snapshots of the heap. We use the following abbreviations.

$$\begin{aligned}
T_0 &= \star \times \star \times \star \\
T_1 &= \texttt{Int} \times \star \times \texttt{Ref}\,T_2 \\
T_2 &= \texttt{Int} \times \texttt{Int} \times \star \\
c &= \texttt{Int}? \times \iota \times (\texttt{Ref}\,T_2)?
\end{aligned}$$

The first line of the program allocates a triple.

$$a_0 \mapsto (42\langle\texttt{Int!}\rangle, 7\langle\texttt{Int!}\rangle, 0\langle\texttt{Int!}\rangle) : T_0$$

The second line sets the third element to be a reference to itself.

$$a_0 \mapsto (42\langle\texttt{Int!}\rangle, 7\langle\texttt{Int!}\rangle, a_0\langle(\texttt{Ref}\,T_0)!\rangle) : T_0$$

The third line casts the reference to $\texttt{Ref}\,T_1$ via (CASTREF1).

$$a_0 \mapsto (42\langle\texttt{Int!}\rangle, 7\langle\texttt{Int!}\rangle, a_0\langle(\texttt{Ref}\,T_0)!\rangle\langle c\rangle : T_1$$

We have a casted value in the heap that needs to be reduced. We apply (HCAST) and (PURECAST) to get

$$a_0 \mapsto (42, 7\langle\texttt{Int!}\rangle, a_0\langle\texttt{Ref}\,T_2\rangle) : T_1$$

We cast address a_0 again, this time to $T_1 \sqcap T_2$, via rule (HDROP) and (CASTREF1).

$$a_0 \mapsto (42, 7\langle\texttt{Int!}\rangle, a_0)\langle\iota \times \texttt{Int}? \times \texttt{Ref}\,T_2\rangle : \texttt{Int} \times \texttt{Int} \times \texttt{Ref}\,T_2$$

A few reductions via (HCAST) and (PURECAST) give us

$$a_0 \mapsto (42, 7, a_0\langle\texttt{Ref}\,T_2\rangle) : \texttt{Int} \times \texttt{Int} \times \texttt{Ref}\,T_2$$

The final cast applied to a_0 is a no-op because the run-time type is already less dynamic than T_2. So we reduce via (HCAST) and (CASTREF2) to:

$$a_0 \mapsto (42, 7, a_0) : \texttt{Int} \times \texttt{Int} \times \texttt{Ref}\,T_2$$

Even though we allow casted values on the heap, we require the normalization of all such casts before returning to the execution of the program. We distinguish between normal heaps of values, μ, and evolving heaps, ν, that may contain both values and casted values. Normal heaps are a subset of the evolving heaps.

4. Type Safety for Monotonic References

We present the high-points of the type safety proof here. The full 26-page formal development and proof is mechanized in Isabelle

2013 and can be found in the supplementary material. The semantics in the mechanized version differs from the semantics presented here in that it uses an abstract machine instead of a reduction semantics, as we found the mechanized proof easier to carry out on an abstract machine. The differences between a reduction semantics and an abstract machine are not important, as one can be derived from the other using the techniques of Biernacka and Danvy (2009).

We begin by lifting the less-dynamic relation to heap typings.

Definition 4 (Less-dynamic relation on heap typings)**.** $\Sigma' \sqsubseteq \Sigma$ *iff* $dom(\Sigma') = dom(\Sigma)$ *and* $\Sigma(a) = T$ *implies* $\Sigma'(a) = T'$ *where* $T' \sqsubseteq T$.

Our first lemma below is important: expression typing is preserved when moving to a less-dynamic heap typing. This allows us to make monotonically-decreasing updates to the heap.

Lemma 5 (Strengthening wrt. the heap typing)**.** *If* $\Gamma; \Sigma \vdash e : T$ *and* $\Sigma' \sqsubseteq \Sigma$*, then* $\Gamma; \Sigma' \vdash e : T$.

Proof sketch. The interesting case is for addresses. We have

$$\frac{\Sigma(a) \sqsubseteq T}{\Gamma; \Sigma \vdash a : T}$$

From $\Sigma' \sqsubseteq \Sigma$ and transitivity of $\sqsubseteq$ (Proposition 1), we have $\Sigma'(a) \sqsubseteq T$. Therefore $\Gamma; \Sigma' \vdash a : T$. □

The definition of well-typed heaps is standard.

Definition 6 (Well-typed heaps)**.** *A heap* ν *is well-typed with respect to heap typing* Σ*, written* $\Sigma \vdash \nu$*, iff* $\forall a\, T.\ \Sigma(a) = T$ *implies* $\nu(a) = cv : T$ *and* $\emptyset; \Sigma \vdash cv : T$ *for some* cv.

From the strengthening lemma, we have the following corollary.

Corollary 7 (Monotonic heap update)**.** *If* $\Sigma \vdash \nu$ *and* $\Sigma(a) = T$ *and* $T' \sqsubseteq T$ *and* $\emptyset; \Sigma \vdash cv : T'$*, then* $\Sigma(a \mapsto T') \vdash \nu(a \mapsto cv : T')$.

Proof sketch. Let $\Sigma' = \Sigma(a \mapsto T')$. From $T' \sqsubseteq T$ we have $\Sigma' \sqsubseteq \Sigma$, so by Lemma 5 we have $\emptyset; \Sigma' \vdash cv : T'$ and $\Sigma' \vdash \nu$. Thus, $\Sigma(a \mapsto T') \vdash \nu(a \mapsto cv : T')$. □

Lemma 8 (Progress and Preservation)**.** *Suppose* $\emptyset; \Sigma \vdash e : T$ *and* $\Sigma \vdash \nu$*. Exactly one of the following holds:*

1. e *is a value, or*
2. $e = \texttt{error}$*, or*
3. $e, \nu \longrightarrow e', \nu'$ *and* $\emptyset; \Sigma' \vdash e' : T$ *and* $\Sigma' \vdash \nu'$ *and* $\Sigma' \sqsubseteq \Sigma$ *for some* e', ν'*, and* Σ'.

Proof sketch. We prove progress and preservation for each of the reduction relations. The most interesting proofs are for the reduction rules that concern casting references. We give the proofs for those cases here.

Case (CastRef1):

$$\frac{\begin{array}{cc}\nu(a) = cv : T_1 & T_3 = T_1 \sqcap T \\ T_3 \neq T_1 & cv' = cv\langle T_1 \Rightarrow T_3 \rangle\end{array}}{a\langle \texttt{Ref}\ T \rangle, \nu \longrightarrow_{cr} a, \nu(a \mapsto cv' : T_3)}$$

Take $e' = a$, $\nu' = \nu(a \mapsto cv' : T_3)$, and $\Sigma' = \Sigma(a \mapsto T_3)$. We have $\Sigma'(a) = T_3$ and $T_3 \sqsubseteq T$ (by Proposition 3), so $\emptyset; \Sigma' \vdash a : T$. Also, from $T_3 \sqsubseteq T$ we have $\Sigma' \sqsubseteq \Sigma$. With $T_3 \sqsubseteq T_1$ (by Proposition 3) we conclude $\Sigma' \vdash \nu'$ by Corollary 7.

Case (CastRef2):

$$\frac{\nu(a) = cv : T_1 \qquad T_1 = T_1 \sqcap T}{a\langle \texttt{Ref}\ T \rangle, \nu \longrightarrow_{cr} a, \nu}$$

Take $e' = a$, $\nu' = \nu$, and $\Sigma' = \Sigma$. We trivially have $\Sigma' \vdash \nu'$, so we just need to show that $\Sigma' \vdash a : T$. From $\Sigma' \vdash \nu'$ and $\nu'(a) = cv : T_1$ we have $\Sigma'(a) = T_1$. From $T_1 = T_1 \sqcap T$ we have $T_1 \sqsubseteq T$ (by Proposition 3). Thus, we conclude that $\Sigma' \vdash a : T$. □

Theorem 9 (Type Safety)**.** *Suppose* $\emptyset; \Sigma \vdash e : T$ *and* $\Sigma \vdash \nu$*. Exactly one of the following holds:*

1. $e, \nu \longrightarrow^* v, \nu'$ *and* $\emptyset; \Sigma' \vdash v : T$ *for some* Σ'*, or*
2. $e, \nu \longrightarrow^* \texttt{error}, \nu'$*, or*
3. e *diverges.*

Proof. If e diverges we immediately conclude the proof. Otherwise, suppose e does not diverge. Then $e, \nu \longrightarrow^* e', \nu'$ and e' cannot reduce. We proceed by induction on the length $e, \nu \longrightarrow^* e', \nu'$, and use Lemma 8 to conclude. □

5. Monotonic References with Blame

We turn to the challenge of designing blame tracking for monotonic references, presenting several examples that motivate and provide intuitions for the design. The later part of this section presents the dynamic semantics of monotonic references with blame tracking.

Consider the following example in which we allocate a reference of dynamic type and then, separately, cast from $\texttt{Ref} \star$ to $\texttt{Ref Int}$ and to $\texttt{Ref Bool}$.

$$\begin{array}{l}\texttt{let}\ r_0 = \texttt{ref}\ (42\ \texttt{as}^{\ell_1}\ \star)\ \texttt{in} \\ \texttt{let}\ r_1 = r_0\ \texttt{as}^{\ell_2}\ \texttt{Ref Int}\ \texttt{in} \\ \texttt{let}\ r_2 = r_0\ \texttt{as}^{\ell_3}\ \texttt{Ref Bool}\ \texttt{in} \\ !r_2\end{array}$$

With monotonic references, the cast at ℓ_3 triggers an error, because $\texttt{Int}$ and $\texttt{Bool}$ are inconsistent. But what blame labels should the error message include? Is it only the fault of ℓ_3? Not really; because ℓ_3 would not cause an error if it were not for the cast at ℓ_2. The casts at ℓ_2 and ℓ_3 disagree with each other regarding the type of the heap cell, so we blame both. The result of this program is $\texttt{blame}\ \{\ell_2, \ell_3\}$.

Next consider an example in which we allocate a reference at type $\texttt{Ref Int}$, cast it to $\texttt{Ref} \star$, and then attempt to write a Boolean.

$$\begin{array}{l}\texttt{let}\ r_0 = \texttt{ref}\ 42\ \texttt{in} \\ \texttt{let}\ r_1 = r_0\ \texttt{as}^{\ell_1}\ \texttt{Ref}\ \star\ \texttt{in} \\ r_1\ :=^{\ell_3}\ (\texttt{true}\ \texttt{as}^{\ell_2}\ \star)\end{array}$$

The update on the third line triggers an error, and we have three possible locations to blame: ℓ_1, ℓ_2, and ℓ_3. The cast at ℓ_2 is from $\texttt{Bool}$ to $\star$, which is harmless. There is no cast at ℓ_3, we are just writing a value of type $\star$ to a reference of type $\texttt{Ref} \star$. The real culprit here is ℓ_1, which casts from $\texttt{Ref Int}$ to $\texttt{Ref} \star$, thereby opening up the potential for the later cast error. Naively, this looks like an upcast, but a proper treatment of subtyping for references makes references invariant. So we have $\texttt{Ref Int} \not<: \texttt{Ref} \star$ and the result of this program is $\texttt{blame}\ \{\ell_1\}$. Figure 9 presents the subtyping relation[1].

We consider a pair of examples below that differ only on the fourth line. We allocate a reference to a pair at type $\texttt{Ref}\ (\star \times \star)$ then cast it to $\texttt{Ref}\ (\texttt{Int} \times \star)$ and to $\texttt{Ref}\ (\star \times \texttt{Int})$. In the first example, we update through the original reference, writing a Boolean and integer, whereas in the second example we write an integer and a

[1] This subtyping relation is for the D variant of blame tracking, and not the more common UD (Siek et al. 2009).

$$\frac{}{B <: B} \qquad \frac{}{T <: \star} \qquad \frac{}{\mathtt{Ref}\ T <: \mathtt{Ref}\ T}$$

$$\frac{T_1' <: T_1 \quad T_2 <: T_2'}{T_1 \to T_2 <: T_1' \to T_2'} \qquad \frac{T_1 <: T_1' \quad T_2 <: T_2'}{T_1 \times T_2 <: T_1' \times T_2'}$$

Figure 9. Subtyping relation

Boolean. Here is the first example:

$$\begin{array}{l}\mathtt{let}\ r_0 = \mathtt{ref}\ (1\ \mathtt{as}^{\ell_1}\ \star, 2\ \mathtt{as}^{\ell_2}\ \star)\ \mathtt{in}\\ \mathtt{let}\ r_1 = r_0\ \mathtt{as}^{\ell_3}\ \mathtt{Ref}\ (\mathtt{Int} \times \star)\ \mathtt{in}\\ \mathtt{let}\ r_2 = r_0\ \mathtt{as}^{\ell_4}\ \mathtt{Ref}\ (\star \times \mathtt{Int})\ \mathtt{in}\\ r_0\ \mathtt{:=}\ (\mathtt{true}\ \mathtt{as}^{\ell_5}\ \star, 2\ \mathtt{as}^{\ell_6}\ \star);\\ \mathtt{fst}\ !r_0\end{array}$$

and here is the second example, just showing the fourth line:

$$\begin{array}{l}\ldots\\ r_0\ \mathtt{:=}\ (1\ \mathtt{as}^{\ell_7}\ \star, \mathtt{true}\ \mathtt{as}^{\ell_8}\ \star);\\ \ldots\end{array}$$

The first example should produce $\mathtt{blame}\,\{\ell_3\}$ while the second example should produce $\mathtt{blame}\,\{\ell_4\}$, but the challenge is how can we associate multiple blame labels with the same heap cell?

We take inspiration from Siek and Wadler (2010) and use *labeled types* for our run-time type information. With a labeled type, each type constructor within the type can be labeled with a type. Figure 10 gives the syntax of labeled types and operations on them, which we shall explain later in this section. In the above examples, the run-time type information for the heap cell evolves in the following way:

$$(\star \times^{\emptyset} \star) \Rightarrow (\mathtt{Int}^{\ell_3} \times^{\emptyset} \star) \Rightarrow (\mathtt{Int}^{\ell_3} \times^{\emptyset} \mathtt{Int}^{\ell_4})$$

In the first example, when we write `true` into the first element of the pair, the cast to `Int` fails and blames ℓ_3, as desired. In the second example, when we write `true` into the second element of the pair, the cast to `Int` fails and blames ℓ_4, as desired.

Our next example brings up a somewhat ambiguous situation. We allocate a reference at type `Ref` $\star$, cast it to `Ref Int` twice, then write a Boolean.

$$\begin{array}{l}\mathtt{let}\ r_0 = \mathtt{ref}\ (42\ \mathtt{as}^{\ell_1}\ \star)\ \mathtt{in}\\ \mathtt{let}\ r_1 = r_0\ \mathtt{as}^{\ell_2}\ \mathtt{Ref\ Int}\ \mathtt{in}\\ \mathtt{let}\ r_2 = r_0\ \mathtt{as}^{\ell_3}\ \mathtt{Ref\ Int}\ \mathtt{in}\\ r_0\ \mathtt{:=}\ (\mathtt{true}\ \mathtt{as}^{\ell_4}\ \star)\end{array}$$

Should we blame ℓ_2 or ℓ_3? In some sense, they are both just as guilty and the ideal would be to blame them both. On the other hand, maintaining potentially large sets of blame labels would induce some space overhead. Our design instead blames the first cast with respect to execution order, in this case ℓ_2.

For our final example, we adapt the above example to have a function in the heap cell so that we can consider the behavior to the left of the arrow.

$$\begin{array}{l}\mathtt{let}\ r_0 = \mathtt{ref}\ (\lambda x{:}\star\,.\,\mathtt{true})\ \mathtt{in}\\ \mathtt{let}\ r_1 = r_0\ \mathtt{as}^{\ell_1}\ \mathtt{Ref}\ (\mathtt{Int} \to \mathtt{Bool})\ \mathtt{in}\\ \mathtt{let}\ r_2 = r_0\ \mathtt{as}^{\ell_2}\ \mathtt{Ref}\ (\mathtt{Int} \to \mathtt{Bool})\ \mathtt{in}\\ r_0\ \mathtt{:=}\ \lambda x{:}\mathtt{Int}.\,\mathtt{zero?}(x);\\ !r_0\ (\mathtt{true}\ \mathtt{as}^{\ell_3}\ \star)\end{array}$$

The run-time type information for the heap cell evolves in the following way:

$$(\star \to^{\emptyset} \mathtt{Bool}^{\emptyset}) \Rightarrow (\mathtt{Int}^{\ell_1} \to^{\emptyset} \mathtt{Bool}^{\emptyset}) \Rightarrow (\mathtt{Int}^{\ell_1} \to^{\emptyset} \mathtt{Bool}^{\emptyset})$$

The function application on the last line of the example triggers a cast error, with the blame going to ℓ_1, again because we wish to blame the first cast with respect to execution order. However, to obtain this semantics some care must be taken. On the second cast, we merge the labeled type for the second cast with the current run-time type information:

$$(\mathtt{Int}^{\ell_1} \to^{\emptyset} \mathtt{Bool}^{\emptyset}) \vartriangle (\mathtt{Int}^{\ell_2} \to^{\emptyset} \mathtt{Bool}^{\emptyset})$$

If we were to use the composition function from Siek and Wadler (2010), the result would be $\mathtt{Int}^{\ell_2} \to^{\emptyset} \mathtt{Bool}^{\emptyset}$ because that composition function is contravariant for function parameters. Here we instead want to be covariant on function parameters, so the result is $\mathtt{Int}^{\ell_1} \to^{\emptyset} \mathtt{Bool}^{\emptyset}$. We define a new function for merging labeled types, $\vartriangle$, in Figure 10.

Semantics of monotonic references with blame Armed with the intuitions from the above examples, we discuss the semantics of monotonic references with blame, defined in Figure 12. The semantics is largely similar to the semantics without blame except that the run-time type information is represented as labeled types and we replace the functions, such as meet ($\sqcap$) that operate on types, with functions such as merge ($\vartriangle$) that operate on labeled types.

Proposition 10 (Meet is the erasure of merge)**.**
If $|P_1| \sim |P_2|$*, then* $|P_1 \vartriangle P_2| = |P_1| \sqcap |P_2|$.
If $|P_1| \not\sim |P_2|$*, then* $P_1 \vartriangle P_2 = \bot^L$ *for some* L.

As discussed with the example above, the definition of $P_1 \vartriangle P_2$ takes into account that P_1 is temporally prior to P_2 and should therefore take precedence with respect to blame responsibility. We use the auxiliary function $p \vartriangle q$ to choose between two optional labels, returning the first if it is present and the second otherwise.

When we cast a reference via rule (6), we need to update the heap cell from labeled type P_1 to P_3. We accomplish this with a new operator $P_1 \Rightarrow P_3$ that produces a coercion. The most interesting line of its definition is for reference types. There we use a different operator, $P \Leftrightarrow Q$, that produces a labeled type and captures the bidirectional read/write nature of mutable references.

The definitions of $\vartriangle$, $\Rightarrow$, and $\Leftrightarrow$ need to percolate errors, which we write as $\bot^L$ where L is a set of blame labels. We use "smart" constructors $\hat{\to}$, $\hat{\times}$, and $\hat{\mathtt{Ref}}$ that return $\bot^L$ if either argument is $\bot^L$ (with precedent to the left if both arguments are errors), but otherwise act like the underlying constructor.

In the rule for allocation, we initialize the RTTI to $T^{\emptyset}$. (Figure 11 defines converting a type to a labeled type.) In the rule for a dynamic dereference, (DynDrfMB), we cast from the reference's run-time labeled type to T by promoting T to the labeled type $T^{\emptyset}$ and then applying the $\Rightarrow$ function to cast between labeled types, so we have $\mu(a)_{\mathsf{rtti}} \Rightarrow T^{\emptyset}$. Suppose that $\mu(a)_{\mathsf{rtti}}$ is $\mathtt{Ref\ Int}^{\ell}$ and T is $\mathtt{Ref}\ \star$. Then the coercion we apply during the dereference is $\mathtt{Int}^{\ell}!$; so our injection coercions contain labeled types. The rule for dynamic update, (DynUpdMB), is dual: we perform the cast $T^{\emptyset} \Rightarrow \mu(a)_{\mathsf{rtti}}$.

Because our injection and projection coercions contain labeled types, the (Collapse) rule becomes

$$v\langle P_1!\rangle\langle P_2?\rangle \longrightarrow_c v\langle P_1 \Rightarrow P_2\rangle \qquad \text{if } |P_1| \sim |P_2|$$

We make similar changes to the (Conflict) rule.

Figure 11 defines the compilation of casts to monotonic coercions. Compared to the compilation without blame (Figure 8), there are three differences. The first two concern injection and projection coercions: instead of only having a blame label on projections we have labeled types inside both injections and projections, as noted above. In the compilation of a cast labeled ℓ, we generate a labeled type for the injection from T by adding the empty label to T, and for the projection to T by adding ℓ to T. The third difference is in the formation of the reference coercion. Instead of simply taking the target type, we use the bidirectional operator $\Leftrightarrow$. Recall the

Optional labels $p, q \quad ::= \quad \emptyset \mid \{\ell\}$
Label sets $L \quad ::= \quad \emptyset \mid \{\ell\} \mid \{\ell_1, \ell_2\}$
Labeled types $P, Q \quad ::= \quad B^p \mid P \to^p P \mid P \times^p P \mid \mathtt{Ref}\ ^p P \mid \star$

Erase labels $\boxed{|P| = T}$

$$|B^p| = B \quad |P \to^p Q| = |P| \to |Q| \quad |P \times^p Q| = |P| \times |Q|$$
$$|\mathtt{Ref}\ ^p P| = \mathtt{Ref}\ |P| \quad |\star| = \star$$

Top label $\boxed{lab(P) = L}$

$$lab(B^p) = p \quad lab(P \to^p Q) = p \quad lab(P \times^p Q) = p$$
$$lab(\mathtt{Ref}\ ^p P) = p \quad lab(\star) = \emptyset$$

Merge optional labels $\boxed{p \triangle p = p}$

$$\{\ell\} \triangle q = \{\ell\} \qquad \emptyset \triangle q = q$$

Merge labeled types $\boxed{P \triangle P = P \text{ or } \bot^L}$

$$B^p \triangle B^q = B^{p \triangle q}$$
$$P \triangle \star = P$$
$$\star \triangle Q = Q$$
$$(P \to^p P') \triangle (Q \to^q Q') = (P \triangle Q) \hat{\to}^{p \triangle q} (P' \triangle Q')$$
$$(P \times^p P') \triangle (Q \times^q Q') = (P \triangle Q) \hat{\times}^{p \triangle q} (P' \triangle Q')$$
$$\mathtt{Ref}\ ^p P \triangle \mathtt{Ref}\ ^q Q = \hat{\mathtt{Ref}}^{\,p \triangle q} (P \triangle Q)$$
$$P \triangle Q = \bot^{lab(P) \cup lab(Q)} \quad \text{otherwise}$$

Bidirectional cast between labeled types $\boxed{P \Leftrightarrow P = P \text{ or } \bot^L}$

$$B^p \Leftrightarrow B^q = B^{\emptyset}$$
$$P \Leftrightarrow \star = P$$
$$\star \Leftrightarrow Q = Q$$
$$(P \to^p P') \Leftrightarrow (Q \to^q Q') = (P \Leftrightarrow Q) \hat{\to}^{\emptyset} (P' \Leftrightarrow Q')$$
$$(P \times^p P') \Leftrightarrow (Q \times^q Q') = (P \Leftrightarrow Q) \hat{\times}^{\emptyset} (P' \Leftrightarrow Q')$$
$$\mathtt{Ref}\ ^p P \Leftrightarrow \mathtt{Ref}\ ^q Q = \hat{\mathtt{Ref}}^{\,\emptyset} (P \Leftrightarrow Q)$$
$$P \Leftrightarrow Q = \bot^{lab(P) \cup lab(Q)} \quad \text{otherwise}$$

Cast between labeled types $\boxed{P \Rightarrow P = c \text{ or } \bot^L}$

$$B^p \Rightarrow B^q = \iota$$
$$\star \Rightarrow \star = \iota$$
$$P \Rightarrow \star = P!$$
$$\star \Rightarrow Q = Q?$$
$$(P \to^p P') \Rightarrow (Q \to^q Q') = (Q \Rightarrow P) \hat{\to} (P' \Rightarrow Q')$$
$$(P \times^p P') \Rightarrow (Q \times^q Q') = (P \Rightarrow Q) \hat{\times} (P' \Rightarrow Q')$$
$$\mathtt{Ref}\ ^p P \Rightarrow \mathtt{Ref}\ ^q Q = \hat{\mathtt{Ref}}\ (P \Leftrightarrow Q)$$
$$P \Rightarrow Q = \bot^{lab(P) \cup lab(Q)} \quad \text{otherwise}$$

Figure 10. Labeled types and their operations

$\boxed{(T \Rightarrow^\ell T) = c}$

$$(B \Rightarrow^\ell B) = \iota$$
$$(\star \Rightarrow^\ell \star) = \iota$$
$$(T \Rightarrow^\ell \star) = T^{\emptyset}!$$
$$(\star \Rightarrow^\ell T) = T^{\ell}?$$
$$(T_1 \to T_2) \Rightarrow^\ell (T_1' \to T_2') = (T_1' \Rightarrow^\ell T_1) \to (T_2 \Rightarrow^\ell T_2')$$
$$(T_1 \times T_2) \Rightarrow^\ell (T_1' \times T_2') = (T_1 \Rightarrow^\ell T_1') \times (T_2 \Rightarrow^\ell T_2')$$
$$\mathtt{Ref}\ T_1 \Rightarrow^\ell \mathtt{Ref}\ T_2 = \mathtt{Ref}\ (T_1^\ell \Leftrightarrow T_2^\ell)$$

Add labels to a type $\boxed{T^\ell = P}$

$$B^\ell = B^\ell \quad (T_1 \to T_2)^\ell = T_1^\ell \to^\ell T_2^\ell \quad (T_1 \times T_2)^\ell = T_1^\ell \times^\ell T_2^\ell$$
$$(\mathtt{Ref}\ T)^\ell = \mathtt{Ref}\ ^\ell T^\ell \quad \star^\ell = \star$$

Figure 11. Compile casts to monotonic coercions (with blame)

second example of this section in which we blamed the cast from $\mathtt{Ref}\ \mathtt{Int}$ to $\mathtt{Ref}\ \star$. By using $\Leftrightarrow$, the resulting coercion is $\mathtt{Ref}\ \mathtt{Int}^{\ell_1}$ instead of $\mathtt{Ref}\ \star$.

6. The Blame Theorem

The blame theorem pin-points the source of cast errors in gradually-typed programs. The blame theorem states that if a program results in a cast error, $\mathtt{blame}\ L$, then the blame labels in L identify the location of implicit casts that did not respect subtyping. Or put positively, the blame labels that occur in safe implicit casts, that is, casts $T_1 \Rightarrow T_2$ where $T_1 <: T_2$, can never be blamed.

We prove the blame theorem via a preservation-style proof (Wadler and Findler 2007, 2009) in which we preserve the $e\ \mathsf{safe}\ \ell$ predicate, a technique due to Siek (2008). This proof will be conducted on the coercion calculus, so to relate the result back to the gradually-typed λ calculus, we need a theorem concerning the relationship between subtyping and coercion blame safety, Theorem 12. Recall that subtyping is defined in Figure 9 and the compilation to coercions is defined in Figure 11. The safe predicate is defined for labeled type, coercions, expressions, and states in Figure 13.

The compilation to coercions relies on the auxiliary function $\Leftrightarrow$ in the case for reference types, so to prove the subtyping and coercion safety theorem, we need the following lemma.

Lemma 11 (Reflexivity of $\Leftrightarrow$ and blame)**.**
For all P and ℓ, $(P \Leftrightarrow P)\ \mathsf{safe}\ \ell$.

Proof. Straightforward by inspection on the definition of $\Leftrightarrow$. □

Theorem 12 (Subtyping and blame safety)**.** *For all T_1, T_2, and ℓ, it holds that $T_1 <: T_2$ iff $(T_1 \Rightarrow^\ell T_2)\ \mathsf{safe}\ \ell$.*

Proof sketch. We prove the forward direction of the implication by induction on the compilation $(T_1 \Rightarrow^l T_2)$. We show only the case for the type $\mathtt{Ref}$, as it relies on the operator $\Leftrightarrow$.
Case $\mathtt{Ref}\ T_1' \Rightarrow^\ell \mathtt{Ref}\ T_2' = \mathtt{Ref}\ (T_1'^\ell \Leftrightarrow T_2'^\ell)$: By definition of $<:$, we have $\mathtt{Ref}\ T_1 <: \mathtt{Ref}\ T_2$ only when $T_1 = T_2$. By Lemma 11 we have that $(T_1^\ell \Leftrightarrow T_1^\ell)\ \mathsf{safe}\ \ell$, and thus $(\mathtt{Ref}\ T_1' \Rightarrow^\ell \mathtt{Ref}\ T_2')\ \mathsf{safe}\ \ell$.

We prove the backward direction of the implication by induction on $(T_1 \Rightarrow^\ell T_2)\ \mathsf{safe}\ \ell$. We show only the case for the type $\to$ for it is contravariant.

Syntax

Expressions	e	$::=$	$\cdots \mid \texttt{blame}\, L$
Coercions	c	$::=$	$\iota \mid P? \mid P! \mid c \to c \mid c \times c \mid c\,;c \mid \texttt{Ref}\, P$
Values	v	$::=$	$k \mid \lambda x.\, e \mid (v, v) \mid v\langle P! \rangle \mid a$
Heap	μ	$::=$	$\emptyset \mid \mu(a \mapsto v : P)$
Evolving Heap	ν	$::=$	$\emptyset \mid \nu(a \mapsto cv : P)$

Coercion typing $\boxed{c : T \Rightarrow T}$

$$\frac{}{P? : \star \Rightarrow |P|} \qquad \frac{}{P! : |P| \Rightarrow \star} \qquad \cdots$$

Pure cast reduction rules $\boxed{e \longrightarrow_c e}$

$$v\langle P_1! \rangle \langle P_2? \rangle \longrightarrow_c v\langle P_1 \Rightarrow P_2 \rangle \quad \text{if } |P_1| \sim |P_2| \quad (\textsc{Collapse})$$

$$v\langle P_1! \rangle \langle P_2? \rangle \longrightarrow_c \texttt{blame}\, L \quad \text{if } P_1 \Rightarrow P_2 = \bot^L \quad (\textsc{Conflict})$$

$$\cdots$$

Cast reduction rules $\boxed{e, \nu \longrightarrow_{cr} e, \nu}$

$$\frac{e \longrightarrow_c e'}{e, \nu \longrightarrow_{cr} e', \nu} \quad (\textsc{PCastB})$$

$$\frac{\nu(a) = cv : P_1 \quad P_3 = P_1 \vartriangle P_2 \quad |P_3| \neq |P_1| \quad cv' = cv\langle P_1 \Rightarrow P_3 \rangle}{a\langle \texttt{Ref}\, P_2 \rangle, \nu \longrightarrow_{cr} a, \nu(a \mapsto cv' : P_3)} \quad (\textsc{CastR1B})$$

$$\frac{\nu(a) = cv : P_1 \quad P_1 = P_1 \vartriangle P_2}{a\langle \texttt{Ref}\, P_2 \rangle, \nu \longrightarrow_{cr} a, \nu} \quad (\textsc{CastR2B})$$

$$\frac{\nu(a) = cv : P_1 \quad P_1 \vartriangle P_2 = \bot^L}{a\langle \texttt{Ref}\, P_2 \rangle, \nu \longrightarrow_{cr} \texttt{blame}\, L, \nu} \quad (\textsc{CastR3B})$$

Program reduction rules $\boxed{e, \mu \longrightarrow_e e, \mu}$

$$\texttt{ref}_T\, v, \mu \longrightarrow_e a, \mu(a \mapsto v : T^\emptyset) \quad \text{if } a \notin dom(\mu)$$

$$!a, \mu \longrightarrow_e \mu(a)_{\mathsf{val}}, \mu \quad (\textsc{DerefMB})$$

$$!a@T, \mu \longrightarrow_e \mu(a)_{\mathsf{val}}\langle \mu(a)_{\mathsf{rtti}} \Rightarrow T^\emptyset \rangle, \mu \quad (\textsc{DynDrfMB})$$

$$a := v, \mu \longrightarrow_e a, \mu(a \mapsto v : \mu(a)_{\mathsf{rtti}}) \quad (\textsc{UpdMB})$$

$$a := v@T, \mu \longrightarrow_e a, \mu(a \mapsto cv : \mu(a)_{\mathsf{rtti}}) \quad (\textsc{DynUpdMB})$$

$$\text{where } cv = v\langle T^\emptyset \Rightarrow \mu(a)_{\mathsf{rtti}} \rangle$$

For $X \in \{cr, e\}$:

$$\frac{e, \nu \longrightarrow_X e', \nu'}{F[e], \nu \longrightarrow_X F[e'], \nu'} \qquad \frac{}{F[\texttt{blame}\, L], \nu \longrightarrow_X \texttt{blame}\, L, \nu}$$

State reduction rules $\boxed{e, \nu \longrightarrow e, \nu}$

$$\frac{e, \mu \longrightarrow_X e', \nu \quad X \in \{cr, e\}}{e, \mu \longrightarrow e', \nu}$$

$$\frac{\nu(a) = cv : P \quad cv, \nu \longrightarrow_{cr} cv', \nu' \quad |\nu'(a)_{\mathsf{rtti}}| = |P|}{e, \nu \longrightarrow e, \nu'(a \mapsto cv' : P)}$$

$$\frac{\nu(a) = cv : P \quad cv, \nu \longrightarrow_{cr} cv', \nu' \quad |\nu'(a)_{\mathsf{rtti}}| \neq |P|}{e, \nu \longrightarrow e, \nu'}$$

$$\frac{\nu(a) = cv : P \quad cv, \nu \longrightarrow_{cr} \texttt{blame}\, L, \nu'}{e, \nu \longrightarrow \texttt{blame}\, L, \nu'}$$

Figure 12. Monotonic references with blame

$\boxed{P\ \mathsf{safe}\ \ell}$

$$\frac{}{\star\ \mathsf{safe}\ \ell} \qquad \frac{\ell \notin p}{K^p\ \mathsf{safe}\ \ell} \qquad \frac{\ell \notin p \quad P_1\ \mathsf{safe}\ \ell \quad P_2\ \mathsf{safe}\ \ell}{P_1 \to^p P_2\ \mathsf{safe}\ \ell}$$

$$\frac{\ell \notin p \quad P_1\ \mathsf{safe}\ \ell \quad P_2\ \mathsf{safe}\ \ell}{P_1 \times^p P_2\ \mathsf{safe}\ \ell} \qquad \frac{\ell \notin p \quad P\ \mathsf{safe}\ \ell}{\texttt{Ref}^{\,p} P\ \mathsf{safe}\ \ell}$$

$\boxed{c\ \mathsf{safe}\ \ell}$

$$\frac{}{\iota\ \mathsf{safe}\ \ell} \qquad \frac{P\ \mathsf{safe}\ \ell}{P?\ \mathsf{safe}\ \ell} \qquad \frac{P\ \mathsf{safe}\ \ell}{P!\ \mathsf{safe}\ \ell}$$

$$\frac{c_1\ \mathsf{safe}\ \ell \quad c_2\ \mathsf{safe}\ \ell}{c_1 \to c_2\ \mathsf{safe}\ \ell} \qquad \frac{c_1\ \mathsf{safe}\ \ell \quad c_2\ \mathsf{safe}\ \ell}{c_1 \times c_2\ \mathsf{safe}\ \ell}$$

$$\frac{c_1\ \mathsf{safe}\ \ell \quad c_2\ \mathsf{safe}\ \ell}{c_1\,;c_2\ \mathsf{safe}\ \ell} \qquad \frac{P\ \mathsf{safe}\ \ell}{\texttt{Ref}\, P\ \mathsf{safe}\ \ell}$$

$\boxed{e\ \mathsf{safe}\ \ell}$

$$\frac{e\ \mathsf{safe}\ \ell \quad c\ \mathsf{safe}\ \ell}{e\langle c \rangle\ \mathsf{safe}\ \ell} \qquad \frac{e\ \mathsf{safe}\ \ell}{\lambda x{:}T.\, e\ \mathsf{safe}\ \ell} \qquad \cdots$$

$\boxed{\nu\ \mathsf{safe}\ \ell}$

$$\frac{\forall a \in dom(\nu).\ \nu(a)_{\mathsf{rtti}}\ \mathsf{safe}\ \ell \text{ and } \nu(a)_{\mathsf{val}}\ \mathsf{safe}\ \ell}{\nu\ \mathsf{safe}\ \ell}$$

$\boxed{e, \nu\ \mathsf{safe}\ \ell}$

$$\frac{e\ \mathsf{safe}\ \ell \quad \nu\ \mathsf{safe}\ \ell}{e, \nu\ \mathsf{safe}\ \ell}$$

Figure 13. Definition of the safety predicate

Case $T_1 \to T_2 <: T_3 \to T_4$: To prove this, we need to derive $(T_3 <: T_1)$ and $(T_2 <: T_4)$. By the induction hypothesis we know that $(T_1 \to T_2) \Rightarrow^\ell (T_3 \to T_4)$ safe ℓ. By definition, this means that $(T_3 \Rightarrow^\ell T_1) \to (T_2 \Rightarrow^\ell T_4)$ safe ℓ. And therefore $(T_3 \Rightarrow^\ell T_1)$ safe ℓ and $(T_2 \Rightarrow^\ell T_4)$ safe ℓ. By induction we thus have $(T_3 <: T_1)$ and $(T_2 <: T_4)$. □

The key to proving the preservation of blame safety for the coercion calculus is showing that our operators on labeled types preserve blame.

Lemma 13 (Blame safety for $\vartriangle$, $\Rightarrow$ and $\Leftrightarrow$)**.** *For all P and Q, if P safe ℓ and Q safe ℓ then $P \oplus Q$ safe ℓ, for $\oplus \in \{\vartriangle, \Rightarrow, \Leftrightarrow\}$.*

Proof. For each operator, we prove blame safety by a straightforward induction. The only non-trivial case is the `Ref` type for $\Rightarrow$, i.e. $\texttt{Ref}^{\,p} P \Rightarrow \texttt{Ref}^{\,q} Q = \hat{\texttt{Ref}}\,(P \Leftrightarrow Q)$, as it relies on the operator $\Leftrightarrow$. In this case we do not appeal to induction but to the blame safety for $\Leftrightarrow$, which can be proved easily as $\Leftrightarrow$ does not rely on other operators. □

Lemma 14 (Preservation of blame safety)**.** *For all e, e', ν, ν', and ℓ, it hold that if e, ν safe ℓ and $e, \nu \longrightarrow e', \nu'$ then e', ν' safe ℓ.*

Proof sketch. We prove blame safety for each of the reduction relations by induction on their derivation. We here show only the most involved cases.
Case (PCastB):

$$\frac{\nu(a) = cv : P_1 \quad P_3 = P_1 \vartriangle P_2 \quad |P_3| \neq |P_1| \quad cv' = cv\langle P_1 \Rightarrow P_3 \rangle}{a\langle \texttt{Ref}\, P_2 \rangle, \nu \longrightarrow_c a, \nu(a \mapsto cv' : P_3)}$$

By assumption we have $\nu\mathsf{safe}\ell$, therefore we can infer $cv\mathsf{safe}\ell$. and $P_1\mathsf{safe}\ell$. By assumption we also have $\langle \texttt{Ref}\, P_2 \rangle\mathsf{safe}\ell$, and therefore

P_2 safe ℓ. Because $P_3 = P_1 \triangle P_2$, by Lemma 13 (Blame Safety for $\triangle$) we infer P_3 safe ℓ. Now, by Lemma 13 (Blame Safety for $\Rightarrow$), we have $P_1 \Rightarrow P_3$ safe ℓ. Therefore $cv' = cv\langle P_1 \Rightarrow P_3 \rangle$ safe ℓ. Now, it is easy to see that $a, \nu(a \mapsto cv' : P_3)$ safe ℓ, as ν safe ℓ by assumption and both cv' safe ℓ and P_3 safe ℓ as inferred above.

(DYNUPDMB):

$$a := v@T, \mu \longrightarrow_e a, \mu(a \mapsto cv : \mu(a)_{\mathsf{rtti}})$$

where $cv = v\langle T^{\emptyset} \Rightarrow \mu(a)_{\mathsf{rtti}} \rangle$. By assumption we have μ, v, and T safe for ℓ. We therefore can infer $\mu(a)_{\mathsf{rtti}}$ safe ℓ. Also, we can apply Lemma 13 (Blame Safety for $\Rightarrow$) to get $T^{\emptyset} \Rightarrow \mu(a)_{\mathsf{rtti}}$ safe ℓ. Therefore $cv = v\langle T^{\emptyset} \Rightarrow \mu(a)_{\mathsf{rtti}} \rangle$ safe ℓ. We finally conclude that $a, \mu(a \mapsto cv : \mu(a)_{\mathsf{rtti}})$ safe ℓ. □

Theorem 15 (Blame Theorem for the coercion calculus)**.** *For all e, ν and ℓ, if e, ν safe ℓ and $e, \emptyset \longrightarrow \mathtt{blame}\ L, \nu$ then $\ell \notin L$.*

Proof. It follows from the preservation of blame safety lemma. □

Definition 16 (Casts for a label in an expression)**.** *Let e be an expression and ℓ a label, we say that e* contains the cast $T_1 \Rightarrow T_2$ for ℓ *whenever, in the derivation of $\Gamma \vdash e \rightsquigarrow e' : T$, there is the creation of a coercion via $T_1 \Rightarrow^{\ell} T_2$.*

Definition 17 (Blame safety for gradually-typed expressions)**.** *A gradually-typed expression e* is safe for ℓ *if all the casts contained in e labeled ℓ respect subtyping.*

Lemma 18 (Translation preserves blame safety)**.** *If e safe ℓ and $\Gamma \vdash e \rightsquigarrow e' : T$, then e' safe ℓ.*

Proof. The proof is a straightforward induction on the derivation of $\Gamma \vdash e \rightsquigarrow e' : T$. We show only the case for function application.

$$\frac{\begin{array}{cc}\Gamma \vdash e_1 \rightsquigarrow e_1' : T_1 & \Gamma \vdash e_2 \rightsquigarrow e_2' : T_2 \\ \mathit{fun}(T_1, T_{11}, T_{12}) & T_2 \sim T_{11} \\ (T_1 \Rightarrow^{\ell'} T_{11} \to T_{12}) = c_1 & (T_2 \Rightarrow^{\ell'} T_{11}) = c_2\end{array}}{\Gamma \vdash (e_1\ e_2)^{\ell'} \rightsquigarrow e_1'\langle c_1 \rangle\ e_2'\langle c_2 \rangle : T_{12}}$$

By the induction hypothesis, we have e_1' safe ℓ and e_2' safe ℓ. By assumption, we have $T_1 <: T_{11} \to T_{12}$ and $T_2 <: T_{11}$. So by Theorem 12, we have c_1 safe ℓ and c_2 safe ℓ. Therefore, we conclude that $e_1'\langle c_1 \rangle\ e_2'\langle c_2 \rangle$ safe ℓ.

□

Theorem 19 (Blame Theorem for the gradually-typed λ calculus with references)**.** *For all e, e', T_1, T_2, ℓ, if*

- $\emptyset \vdash e \rightsquigarrow e' : T$,
- e safe ℓ,
- $e', \emptyset \longrightarrow \mathtt{blame}\ L, \nu$

then $\ell \notin L$.

Proof. From the assumptions we have e' safe ℓ by Lemma 18. Then we conclude by applying the Blame Theorem for the coercion calculus. □

7. Implementation concerns wrt. strong updates

The monotonic semantics for references performs in-place updates to the heap with values of different type. In languages where values have uniform type, like many functional and object-oriented languages, this does not pose a problem. However, for languages where values may have different sizes, in-place updates do pose a problem. We recently discovered a solution inspired by garbage collection techniques. When the semantics says to do an in-place update with a larger value, what the implementation can do is allocate a new piece of memory and place a forwarding pointer in the old location. When reading and writing through dynamic references, the implementation would need to check for and follow the forwarding pointers. However, when reading and writing through fully-static references, the implementation would not need to worry about forwarding pointers because a fully-static heap cell is never moved. Also, during a garbage collection the implementation could collapse sequences of forwarding pointers to reduce the overhead of them in the subsequent execution.

8. Related Work

Interest in integrating static and dynamic typing within the same language has existed for some time, and early approaches included the addition of a `dynamic` type to a static type system (Abadi et al. 1989) as well as the quasi-static typing of Thatte (1990).

Siek and Taha (2006) introduced the term "gradual typing" to describe such systems, and were the first to study the interaction between gradual typing and mutable references, but used an inflexible consistency relation that did not allow implicit casts between references of different type. Herman et al. (2007, 2010) relaxed the consistency rule for references and introduced the standard semantics, in which casting a reference creates a proxied reference that performs casts on every read and write.

Findler and Felleisen (2002) introduced the technique of blame tracking to provide debugging information when constraints are violated. This work was developed in the context of contracts, but it was adapted to casts in gradual type systems by Wadler and Findler (2007, 2009), who also formulated the blame theorem. Dimoulas et al. (2012) proved the blame theorem for the standard semantics of Herman et al. (2007).

Henglein (1994) developed the coercion calculus to aid in understanding the compilation of dynamically-typed languages to statically typed ones. Herman et al. (2007, 2010) used it to design a space-efficient approach to casts in gradually-typed languages. Siek et al. (2009) discussed several approaches to integrating blame tracking into the coercion calculus, and Siek and Wadler (2010) introduced threesomes, which similarly can represent sequences of casts in a space-efficient manner with blame tracking.

The casts and coercions studied in this paper bear many similarities with contracts (Findler and Felleisen 2002). Racket (Flatt and PLT 2014) provides contracts for mutable values in the form of impersonators (Strickland et al. 2012), which, for our purposes, can be viewed as implementing the standard semantics of Herman et al. (2007), as we saw in Section 2.

Wrigstad et al. (2010) develop an alternative system that avoids proxies in typed code by instead introducing a distinction between concrete types, which are only inhabited by one kind of value, and *like types*, for which there can be multiple kinds of values. However, their approach did not support blame tracking, it restricts the flow of values between dynamic code and concretely-typed code, and like types add complexity from the programmer's viewpoint.

Fähndrich and Leino (2003) introduce a technique similar to monotonic references with their monotonic typestate. In this design, objects may flow from less restrictive to more restrictive typestates, but not vice versa. Unlike monotonic references, which require runtime checks due to the existence of dynamically-typed regions of code, in their system monotonicity is enforced statically.

Swamy et al. (2014) design a gradually-typed variant of JavaScript named TS*, that is type safe despite interacting with untrusted JavaScript contexts that can walk the stack, use `eval`, and perform prototype poisoning. They use an RTTI-based approach to cast objects, similar to the way we associate RTTI with references. Swamy et al. (2014) allow their heap to evolve monotonically, but with respect to subtyping instead of the less-dynamic relation as

we do. The advantage of using subtyping is that dereference and update can never trigger cast errors. On the other hand, their approach still requires run-time overhead, even in fully-static code. For example, a reference of type $\texttt{Ref}\ (\texttt{Int} \to \texttt{Int})$ may point to a function of type $\star \to \texttt{Int}$ because $\star \to \texttt{Int} <: \texttt{Int} \to \texttt{Int}$, and so a dereference in fully-static code has to perform a cast. Finally, TS* does not perform blame tracking whereas we have showed how to perform blame tracking for monotonic references and proved the blame theorem.

Gradual typing was added to C$^\sharp$ with the addition of the `dynamic` type (Hejlsberg 2010). Bierman et al. (2010) define a formal model of C$^\sharp$, named FC$^\sharp_4$, and present an operational semantics. The semantics is similar to that of Swamy et al. (2014) in that they use an RTTI-based approach and subtype checks to implement casts.

Many gradually-typed systems sidestep this issue entirely by employing a type-erasure semantics, that is, they implement a gradual type checker but do not insert run-time casts (Hejlsberg 2012). The advantage of the type-erase semantics is its ease of implementation and lack of run-time overhead (beyond the normal amount for a dynamically-typed language), but the disadvantage is that one gives up a static notion of type soundness and there remains run-time overhead in fully-static code.

9. Conclusion

We have presented a new design for gradually-typed mutable references, called monotonic references, which is the first to incur zero-overhead for reference accesses in statically typed code while maintaining the full expressiveness of gradual typing. Further, the design does not use reference proxies, so it does not disrupt object identity. We defined a dynamic semantics for monotonic references and presented a mechanized proof of type safety. Further, we defined a blame tracking strategy based on using labeled types in the run-time type information and proved the blame theorem.

References

M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Symposium on Principles of programming languages*, 1989.

Anonymous. Blame, coercion, and threesomes: Together again for the first time. In submission., 2014.

G. Bierman, E. Meijer, and M. Torgersen. Adding dynamic types to C#. In *European Conference on Object-Oriented Programming*, ECOOP'10. Springer-Verlag, 2010.

M. Biernacka and O. Danvy. Towards compatible and interderivable semantic specifications for the Scheme programming language, Part II: Reduction semantics and abstract machines. In *Semantics and Algebraic Specification: Essays dedicated to Peter D. Mosses on the occasion of his 60th birthday*, pages 186–206, 2009.

B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn: Robust, concurrent, extensible scripting on the jvm. In *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 117–136, 2009.

G. Bracha. *Optional Types in Dart*. Google, October 2011.

C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Complete monitors for behavioral contracts. In *ESOP*, 2012.

M. Fähndrich and K. R. M. Leino. Heap monotonic typestate. In *International Workshop on Alias Confinement and Ownership (IWACO)*, July 2003.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, ICFP, pages 48–59, October 2002.

M. Flatt and PLT. The Racket reference 6.0. Technical report, PLT Inc., 2014. `http://docs.racket-lang.org/reference/index.html`.

A. Hejlsberg. C# 4.0 and beyond by anders hejlsberg. Microsoft Channel 9 Blog, April 2010.

A. Hejlsberg. Introducing TypeScript. Microsoft Channel 9 Blog, October 2012.

F. Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.

D. Herman and C. Flanagan. Status report: specifying JavaScript with ML. In *ML '07: Proceedings of the 2007 workshop on Workshop on ML*, pages 47–52. ACM, 2007. ISBN 978-1-59593-676-9.

D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, page XXVIII, April 2007.

D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167–189, 2010.

A. Oliart. An algorithm for inferring quasi-static types. Technical Report 1994-013, Boston University, 1994.

J. G. Siek. A theory of gradual typing (draft). September 2008.

J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

J. G. Siek and W. Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, volume 4609 of *LCNS*, pages 2–27, August 2007.

J. G. Siek and P. Wadler. Threesomes, with and without blame. In *Symposium on Principles of Programming Languages*, POPL, pages 365–376, January 2010.

J. G. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In *European Symposium on Programming*, ESOP, pages 17–31, March 2009.

G. L. Steele, Jr. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990. ISBN 1-55558-041-6.

T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In *Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, 2012.

N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. Bierman. Gradual typing embedded securely in javascript. In *ACM Conference on Principles of Programming Languages (POPL)*, January 2014.

S. Thatte. Quasi-static typing. In *POPL 1990*, pages 367–381, New York, NY, USA, 1990. ACM Press. ISBN 0-89791-343-4.

S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *OOPSLA'06 Companion*, pages 964–974, NY, 2006. ACM.

S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Symposium on Principles of Programming Languages*, January 2008.

T. Van Cutsem and M. S. Miller. Proxies: Design principles for robust object-oriented intercession apis. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, pages 59–72, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0405-4. . URL `http://doi.acm.org/10.1145/1869631.1869638`.

J. Verlaguet. Facebook: Analyzing PHP statically. In *Commercial Users of Functional Programming (CUFP)*, 2013.

M. M. Vitousek, S. Bharadwaj, and J. G. Siek. Towards gradual typing in Jython. In *Scripts to Programs Workshop (STOP)*, 2012.

M. M. Vitousek, J. G. Siek, A. Kent, and J. Baker. Design and evaluation of gradual typing for Python. (draft), June 2014.

P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Workshop on Scheme and Functional Programming*, pages 15–26, 2007.

P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, ESOP, pages 1–16, March 2009.

T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *Symposium on Principles of Programming Languages*, POPL, pages 377–388, January 2010.