Monotonic References for Efficient Gradual Typing

Jeremy G. Siek[1], Michael M. Vitousek[1], Matteo Cimini[1], Sam Tobin-Hochstadt[1], and Ronald Garcia[2]

[1] Indiana University Bloomington
jsiek@indiana.edu
[2] 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 regions between the two typing disciplines. One goal of gradual typing is to provide all the benefits of static typing, such as efficiency, in statically-typed regions. However, this goal is elusive: the standard approach to mutable references imposes run-time overhead in statically-typed regions and alternative approaches are too conservative, either statically or at run-time. In this paper we present a new semantics called *monotonic references* which imposes none of the run-time overhead of dynamic typing in statically typed regions. With this design, casting a reference may cause a heap cell to become more statically typed (but not less). Retaining type safety is challenging with strong updates to the heap. Nevertheless, we have a mechanized proof of type safety. Further, we present blame tracking for monotonic references and prove a blame theorem.

1 Introduction

Static and dynamic type systems have well-known strengths and weaknesses. Static type systems provide machine-checked documentation, catch bugs early, and enable efficient code. Dynamic type systems provide the flexibility often needed during prototyping and enable powerful features such as reflection. Over the years, many languages blurred the boundary between static and dynamic typing, such as type hints in Lisp and the addition of a dynamic type to otherwise statically typed languages [Abadi et al., 1989]. But the seamless and sound integration of static and dynamic typing remained problematic until two pieces fell into place: the gradual type system of Siek and Taha [2006] and the blame theorems of Tobin-Hochstadt and Felleisen [2006] and Wadler and Findler [2009].

However, there have been challenges regarding the efficiency of gradual typing. One issue concerns the efficiency of mutable references in statically-typed code. Consider the following statically-typed function f that dereferences its parameter x.

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

In the first application of f, a normal reference to an integer flows into f. For the second application, we allocate a reference of type $\texttt{Ref}\,\star$ ($\star$ is the dynamic type) then implicitly cast it to $\texttt{Ref Int}$ before applying f. According to the standard semantics of Herman et al. [2007], this cast wraps the reference in a proxy. Thus code generated for the dereference in the body of f must inspect the reference to find out whether it is a normal reference or a proxied reference, and in the proxied case, apply a coercion.

Before discussing solutions to this problem, we recall an important property of the standard semantics for mutable references, and of gradual typing in general. Removing type annotations does not affect the behavior of a program: it should still type check and the result should be the same modulo proxies. (Adding type annotations, on the other hand can sometimes induce static type errors and cast errors.) Consider the statically-typed program on the left that increments 41 in a way that exercises references and functions. In the code on the right we remove the type annotation on y, but the program still type checks and the result remains 42.

$$\begin{array}{lcl}
\texttt{let}\,f = \texttt{ref}\,(\lambda y : \texttt{Int}\,.\,y + 1)\;\texttt{in} & & \texttt{let}\,f = \texttt{ref}\,(\lambda y.\,y + 1)\;\texttt{in} \\
\texttt{let}\,g = \lambda h{:}\texttt{Ref}\,(\texttt{Int}{\to}\texttt{Int}).\,!h\;41 & \Longrightarrow & \texttt{let}\,g = \lambda h{:}\texttt{Ref}\,(\texttt{Int}{\to}\texttt{Int}).\,!h\;41 \\
\texttt{in}\;\;g(f) & & \texttt{in}\;\;g(f)
\end{array}$$

Wrigstad et al. [2010] address the efficiency problem by introducing 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, may refer to dynamically-typed values but incur run-time overhead. The distinction between like types and concrete types achieves the efficiency goals, but the restrictions in their type system mean that removing concrete type annotations, as in the above example, can trigger a static type error.

In this paper we investigate the essence of the run-time overhead problem in the context of the gradually-typed lambda calculus with mutable references. We propose a semantics, *monotonic references* that enables the compilation of statically-typed regions to machine code that is free of any of the indirection or run-time checking associated with dynamic typing, like boxing or bit tags. Monotonic references allow dynamically-typed values to flow into code with (concrete) static types. When a reference flows through a cast, the cast may coerce its underlying heap cell to become more statically typed. In general, this means that values in the heap may evolve monotonically with respect to the precision relation (Section 2).

Monotonic references preserve a global invariant that a value in the heap is at least as precise as any reference that points to it. Thus, a fully-static reference always points to a value of the same type, so there is no overhead associated with reading or writing through the reference: the reads and writes may be implemented as machine loads and stores. By a "fully static reference" we mean that there are no occurrences of the dynamic type $\star$ in the pointed-to type of the reference, such as $\texttt{Ref Int}$ and $\texttt{Ref}\,(\texttt{Int} \times \texttt{Bool})$. Reads and writes to references that are not fully-static, such as $\texttt{Ref}\,\star$ and $\texttt{Ref}\,(\star\times\texttt{Bool})$, still require casts: the dynamic regions have to pay their own way.

Swamy et al. [2014] and Rastogi et al. [2014] integrate static and dynamic typing in the context of TypeScript with the TS* and Safe TypeScript languages. Both use a notion of monotonicity in the heap, but with respect to subtyping, treating $\star$ as a universal supertype, instead of with respect to the precision relation. Because these languages compile to JavaScript, they inherit the overhead of dynamic typing, whereas with monotonic references, the overhead of dynamic typing occurs only in dynamically-typed code. In the example above, removing the type annotation from y causes TS* to halt the program with a cast error at the implicit cast from $\texttt{Ref}\,(\star{\rightarrow}\star)$ to $\texttt{Ref}\,(\texttt{Int}{\rightarrow}\texttt{Int})$. TS* only allows up-casts on functions, but $\star{\rightarrow}\star \not<: \texttt{Int}{\rightarrow}\texttt{Int}$. In contrast, the monotonic references design allows interoperation between static and dynamic code, so long as the dynamic values are consistent with the static types. TS* provides valuable security guarantees without incurring much more overhead. Safe TypeScript further reduces the overheads through the use of differential coercive subtyping and by stratifying types in a way that is similar to like types.

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. Blame tracking also enables fine-grained guarantees, via a blame theorem, regarding which regions of the code are statically type safe. In this paper we present blame tracking for monotonic references and prove a blame theorem. 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 between labeled types.

To summarize, this paper presents a new semantics for gradually-typed mutable references that delivers guaranteed efficiency for the statically-typed parts of a program, maintains type safety, and provides blame tracking, while continuing to enable fine-grained migration between static and dynamic code, even in the presence of higher-order functions. This paper makes the following technical contributions:

1. We define the semantics of monotonic references (Sections 3 and 5).
2. We discuss our proof of type safety, mechanized in Isabelle (Section 4).
3. We augment monotonic references with blame tracking and prove the blame-subtyping theorem (Section 6).

We review the gradually-typed lambda calculus with references in Section 2 and discuss the run-time overhead associated with mutable references. We address an implementation concern regarding strong updates in Section 7. The paper concludes in Section 9.

2 Background and Problem Statement

Figure 1 reviews the syntax and static semantics of the gradually-typed lambda calculus with references. The primary difference between gradual typing and static typing is that uses of type equality are replaced with *consistency* (aka. compatibility), also defined in Figure 1. The consistency relation enables implicit casts to and from $\star$. (In contrast, an object-oriented language only allows implicit casts to the top `Object` type.) This consistency relation is a congruence, even for reference types [Herman et al., 2007], which differs from the original treatment of references as invariant [Siek and Taha, 2006]. The more flexible treatment of references enables the passing of references between more and less dynamically typed regions of code, but is also the source of the difficulties that we solve in this paper. The precision relation, which says whether one type is more or less dynamic than another, is also defined in Figure 1, and is closely related to consistency. Two types are consistent when there exists a greatest lower bound with respect to the precision relation. This relation is also known as naïve subtyping [Wadler and Findler, 2009].

All of the types, except for $\star$, classify unboxed values. So, for example, `Int` is the type for native integers (e.g. 64-bit integers). The auxiliary relations *fun*, *pair*, and *ref*, defined in Figure 1, implement pattern matching on types, enabling a more concise presentation of the typing rules compared to prior presentations of gradual type systems. Labels ℓ represent source code locations that would be captured during parsing.

The dynamic semantics of the gradually-typed lambda calculus is defined by a type-directed translation to the coercion calculus [Henglein, 1994], using the standard semantics for mutable references originally due to Herman et al. [2007].

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 2 gives the compilation of casts into coercions, written $(T \Rightarrow^{\ell} T) = c$. The compilation of gradually-typed terms into the coercion-based calculus is otherwise straightforward, so we give just the function application rule as an example:

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

Figures 3 and 4 define the coercion-based calculus. 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 run-time overhead problem in the next subsection. For an introduction to the coercion calculus, we refer to Henglein [1994].

Syntax

$$
\begin{array}{lll}
\text{Base types} & B & ::= \texttt{Int} \mid \texttt{Bool} \\
\text{Types} & T & ::= B \mid T \to T \mid T \times T \mid \texttt{Ref}\, T \mid \star \\
\text{Labels} & \ell & \\
\text{Operators} & op & ::= \texttt{plus} \mid \texttt{minus} \mid \texttt{is} \mid \cdots \\
\text{Expressions} & e & ::= k \mid op^{\ell}(\vec{e}) \mid x \mid \lambda x{:}T.\, e \mid (e\ e)^{\ell} \mid e\ \texttt{as}^{\ell}\ T \mid \\
 & & \quad (e, e) \mid \texttt{fst}^{\,\ell} e \mid \texttt{snd}^{\,\ell} e \mid \texttt{ref}\, e \mid \texttt{!}^{\ell} e \mid e\ \texttt{:=}^{\ell}\ e
\end{array}
$$

$$\lambda x.\, e \equiv \lambda x{:}\,\star\,.\, e$$

Consistency $\boxed{T \sim T}$

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

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

Precision $\boxed{T \sqsubseteq T}$

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

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

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

$$
\frac{k : B}{\Gamma \vdash k : B} \qquad \frac{\Gamma \vdash \vec{e} : \vec{T} \qquad op : \vec{B} \to B \qquad \vec{T} \sim \vec{B}}{\Gamma \vdash op^{\ell}(\vec{e}) : B} \qquad \frac{\Gamma \vdash e : T_1 \qquad T_1 \sim T_2}{\Gamma \vdash e\ \texttt{as}^{\ell}\ T_2 : T_2}
$$

$$
\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\Gamma(x \mapsto T_1) \vdash e : T_2}{\Gamma \vdash \lambda x{:}T_1.\, e : T_1 \to T_2} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \\ \mathit{fun}(T_1, T_{11}, T_{12}) \qquad T_2 \sim T_{11}\end{array}}{\Gamma \vdash (e_1\ e_2)^{\ell} : T_{12}}
$$

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

$$
\frac{\Gamma \vdash e : T}{\Gamma \vdash \texttt{ref}\, e : \texttt{Ref}\, T} \qquad \frac{\Gamma \vdash e : T \qquad \mathit{ref}(T, T')}{\Gamma \vdash \texttt{!}^{\ell} e : T'} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \\ \mathit{ref}(T_1, T_1') \qquad T_2 \sim T_1'\end{array}}{\Gamma \vdash e_1\ \texttt{:=}^{\ell}\ e_2 : T_1}
$$

Type matching

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

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

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

Fig. 1. Gradually-typed λ calculus with mutable references

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

$$\begin{array}{lcllcl} (B \Rightarrow^{\ell} B) & = & \iota & (I \Rightarrow^{\ell} \star) & = & I! \\ (\star \Rightarrow^{\ell} \star) & = & \iota & (\star \Rightarrow^{\ell} I) & = & I?^{\ell} \end{array}$$

$$\begin{array}{rcl} (T_1{\to}T_2) \Rightarrow^{\ell} (T_1'{\to}T_2') & = & (T_1' \Rightarrow^{\ell} T_1){\to}(T_2 \Rightarrow^{\ell} T_2') \\ (T_1 \times T_2) \Rightarrow^{\ell} (T_1' \times T_2') & = & (T_1 \Rightarrow^{\ell} T_1') \times (T_2 \Rightarrow^{\ell} T_2') \\ \texttt{Ref}\,T \Rightarrow^{\ell} \texttt{Ref}\,T' & = & \texttt{Ref}\,(T \Rightarrow^{\ell} T')\,(T' \Rightarrow^{\ell} T) \end{array}$$

Fig. 2. Compile casts to coercions

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

Fig. 3. Syntax for the coercion-based calculus with mutable references

2.1 Run-time overhead in fully-static code

Recall the example in Section 1 in which the dereference of a statically-typed reference must first check whether the reference is proxied or not.

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

The overhead can be seen in the dynamic semantics (Figure 4), 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 eliminate this 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.

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 $\boxed{e, \mu \longrightarrow_r e, \mu}$

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

State reduction rules

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

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

Fig. 4. Coercion-based calculus with mutable references

2.2 Non-determinism in multi-threaded code

This standard semantics for mutable references produces an error only if type inconsistency is witnessed by some read or write to a particular reference, so in a non-deterministic multi-threaded program, whether a check will fail at run-time is difficult to predict.

The contract system in Racket implements the standard semantics [Flatt and PLT, 2014]. For example, the following program sometimes fails and blames `b1`, sometimes fails and blames `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 heap 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

Figures 5 and 6 define 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. Typical of gradually-typed languages, there is a value form for values that have been boxed and injected to $\star$, which is $v\langle I!\rangle$. The I plays the role of a tag that records the type of v. The values at all other types are unboxed, as they would be in a statically-typed language.

With monotonic references, only one kind of value has 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 more precise (Figure 1). Otherwise the cast results in a run-time error. Thus, we maintain the heap invariant that the type of each reference in the program is less or equally precise as the type of the value on the heap that it points to, as captured in the typing rule (WTRef).

One might wonder why our heap invariant uses the precision relation instead of subtyping. Could we obtain the same efficiency goals using subtyping instead? Consider the following program in which a function of type $\star \to \mathtt{Int}$ is referenced from the static type $\mathtt{Int} \to \mathtt{Int}$. (We have $\star \to \mathtt{Int} <: \mathtt{Int} \to \mathtt{Int}$.)

$$\begin{array}{l}\mathtt{let}\, r_1 = \mathtt{ref}\,(\lambda x : \star.\ x)\ \mathtt{in} \\ \mathtt{let}\, r_2 = (r_1\ \mathtt{as}\ \mathtt{Ref}\,(\mathtt{Int} \to \mathtt{Int}))\ \mathtt{in} \\ !r_2\ 42\end{array}$$

The dereference of r_2 should not require overhead, but we have a function of type $\star \to \mathtt{Int}$ that is to be applied to an integer, and the conversion from $\mathtt{Int}$ to $\star$ requires boxing in our setting. Thus, the dereference of r_2 is not simply a load instruction, but it must handle the casting from $\star \to \mathtt{Int}$ to $\mathtt{Int} \to \mathtt{Int}$. In general, given a reference of type $\mathtt{Ref}\, T_2$, even when T_2 is a fully-static type, there are many types T_1 such that $T_1 <: T_2$ and $T_1 \neq T_2$. (In TS*, the dereference does not need to cast the function, but that is because integers are always boxed, which induces overhead elsewhere.)

The syntax of the monotonic calculus differs from the standard calculus in that there are two kinds of dereference and update expressions. Programmers need not worry about choosing which of the two dereference or update expressions to use because this choice is type-directed and therefore is handled during compilation from the source language to the coercion calculus. We reserve the forms $!e$ and $e_1 := e_2$ for situations in which the reference type is fully static (Figure 14 and expression typing in Figure 6). In these situations we know that the value in the heap has the same type as the reference. Thus, if a reference has a fully static type, such as $\mathtt{Ref}\ \mathtt{Int}$, 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 $\mathtt{Ref}\, T$. For example, T could be $\star$, $\star \times \star$, or $\star \times \mathtt{Int}$. Because the value on the heap might be more precise 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 incoming value v from T to the address's run-time type, so the new content of the cell

$$
\begin{array}{lll}
\text{Expressions} & e & ::= \;.. \mid \texttt{ref}_T\, e \mid \texttt{!}e@T \mid e\;\texttt{:=}\;e@T \mid \texttt{error} \\
\text{Coercions} & c & ::= \iota \mid I? \mid I! \mid c{\to}c \mid c \times c \mid c\,;c \mid \texttt{Ref}\,T \\
\text{Values} & v & ::= k \mid \lambda x.\,e \mid (v,v) \mid v\langle I!\rangle \mid a \\
\text{Casted Values} & cv & ::= v \mid v\langle c\rangle \mid (cv,cv) \\
\text{Heap} & \mu & ::= \emptyset \mid \mu(a \mapsto v : T) \\
\text{Evolving Heap} & \nu & ::= \emptyset \mid \nu(a \mapsto cv : T) \\
\text{Frames} & F & ::= \;.. \mid \texttt{!}\square@T \mid \square\;\texttt{:=}\;e@T \mid v\;\texttt{:=}\;\square@T
\end{array}
$$

Fig. 5. Syntax for monotonic references without blame

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 no more dynamic than (i.e. at least as static as) 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).

The rest of the coercion reduction rules are captured by the rule (PureCast), so they are the same as in the standard semantics (Figure 4), 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 computes the greatest lower bound with respect to the precision relation.

To motivate our organization of the heap, we present two examples that demonstrate why we store run-time type information and casted values, 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.

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

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

$$\frac{\begin{array}{c}\Gamma; \Sigma \vdash e : \texttt{Ref}\,T \\ \mathit{static}\ T\end{array}}{\Gamma; \Sigma \vdash \texttt{!}e : T} \qquad \frac{\begin{array}{c}\Gamma; \Sigma \vdash e_1 : \texttt{Ref}\,T \\ \Gamma; \Sigma \vdash e_2 : T \quad \mathit{static}\ T\end{array}}{\Gamma; \Sigma \vdash e_1 \texttt{ := } e_2 : \texttt{Ref}\,T} \qquad \frac{\Gamma; \Sigma \vdash e : \texttt{Ref}\,T}{\Gamma; \Sigma \vdash \texttt{!}e@T : T}$$

$$\frac{\Gamma; \Sigma \vdash e_1 : \texttt{Ref}\,T \quad \Gamma; \Sigma \vdash e_2 : T}{\Gamma; \Sigma \vdash e_1 \texttt{ := } e_2@T : \texttt{Ref}\,T} \quad \dots \quad \frac{\Sigma(a) \sqsubseteq T_2}{\Gamma; \Sigma \vdash a : T_2} \qquad \text{(WTRef)}$$

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

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

$$\frac{\begin{array}{c}\nu(a) = cv : T_1 \quad T_3 = T_1 \sqcap T_2 \\ T_3 \neq T_1 \quad cv' = cv\langle T_1 \Rightarrow T_3\rangle\end{array}}{a\langle \texttt{Ref}\,T_2\rangle, \nu \longrightarrow_{cr} a, \nu(a \mapsto cv' : T_3)} \qquad \text{(CastRef1)}$$

$$\frac{\nu(a) = cv : T_1 \quad T_1 = T_1 \sqcap T_2}{a\langle \texttt{Ref}\,T_2\rangle, \nu \longrightarrow_{cr} a, \nu} \qquad \text{(CastRef2)}$$

$$\frac{\nu(a) = cv : T_1 \quad T_1 \not\sim T_2}{a\langle \texttt{Ref}\,T_2\rangle, \nu \longrightarrow_{cr} \texttt{error}, \nu} \qquad \text{(CastRef3)}$$

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

$$\begin{array}{rll}
e, \mu \longrightarrow_e & e', \mu \quad \text{if } e \longrightarrow e' & \\
\texttt{ref}_T\, v, \mu \longrightarrow_e & a, \mu(a \mapsto v : T) \quad \text{if } a \notin \mathit{dom}(\mu) & \\
\texttt{!}a, \mu \longrightarrow_e & \mu(a)_{\mathsf{val}}, \mu & \text{(DerefM)} \\
\texttt{!}a@T, \mu \longrightarrow_e & \mu(a)_{\mathsf{val}}\langle \mu(a)_{\mathsf{rtti}} \Rightarrow T\rangle, \mu & \text{(DynDerefM)} \\
a \texttt{ := } v, \mu \longrightarrow_e & a, \mu(a \mapsto v : \mu(a)_{\mathsf{rtti}}) & \text{(UpdM)} \\
a \texttt{ := } v@T, \mu \longrightarrow_e & a, \mu(a \mapsto cv : \mu(a)_{\mathsf{rtti}}) & \text{(DynUpdM)} \\
& \text{where } cv = v\langle T \Rightarrow \mu(a)_{\mathsf{rtti}}\rangle &
\end{array}$$

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 $\boxed{e, \nu \longrightarrow e, \nu}$

$$\frac{e, \mu \longrightarrow_X e', \nu \quad X \in \{cr, e\}}{e, \mu \longrightarrow e', \nu} \qquad \frac{\begin{array}{c}\nu(a) = cv : T \quad cv, \nu \longrightarrow_{cr} cv', \nu' \\ \nu'(a)_{\mathsf{rtti}} = T\end{array}}{e, \nu \longrightarrow e, \nu'(a \mapsto cv' : T)} \quad \text{(HCast)}$$

$$\frac{\nu(a) = cv : T \quad cv, \nu \longrightarrow_{cr} \texttt{error}, \nu'}{e, \nu \longrightarrow \texttt{error}, \nu'} \qquad \frac{\begin{array}{c}\nu(a) = cv : T \quad cv, \nu \longrightarrow_{cr} cv', \nu' \\ \nu'(a)_{\mathsf{rtti}} \neq T\end{array}}{e, \nu \longrightarrow e, \nu'}$$

(HDrop)

Fig. 6. Monotonic references without blame

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

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

Fig. 7. The meet function (greatest lower bound)

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 naïve 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.

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

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

There are several 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 more precise 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.

Encoding permissive references The monotonic discipline and its run-time invariant-enforcement seems to restrict how developers can formulate their programs. It is natural to ask whether monotonic references are compatible with the flexibility that is expected in dynamic languages. In this section we show that the monotonic discipline admits permissive references through a syntactic discipline that can be conveniently provided to programmers.

Consider the following program that uses an allocated reference cell at two incompatible types, `Int` and `Bool`.

```
let x = ref (4 as ⋆) in
let y = (x as Ref Int) in
let z = (x as Ref Bool) in
!y;
z := true;
!z
```

Under the standard reference semantics, this program runs without incident, but under monotonic references it fails just as the cell is updated to `true`. We can regain this flexibility under monotonic references via a disciplined use of $\star$ typed reference cells. Consider the following rewrite of this program:

```
let x = ref (4 as ⋆) in
let y = x in                    // treat y like Ref Int
let z = x in                    // treat z like Ref Bool
(!y) as Int;
(z := (true) as ⋆) as Bool;
(!z) as Bool
```

In this encoding, all references have type $\texttt{Ref}\,\star$, and typing is enforced only at dereferences and updates, using ascriptions. This program runs successfully under the monotonic semantics, but it would be tedious and error prone to insert these ascriptions by hand.

Luckily there is no need: we codify this permissive reference discipline by introducing a surface language that makes this convenient. We extend the expressions with *permissive references* $\widetilde{\texttt{ref}}\,e$, and the types with a corresponding type $\widetilde{\texttt{Ref}}\,T$. Consistency is extended so that permissive references have the same consistency properties as monotonic references, but permissive references are not consistent with monotonic references.

Finally we introduce a type-directed transformation $\Gamma \vdash e : T \rightsquigarrow e$ that translates permissive references to monotonic references. The interesting cases are presented below.

$$\frac{x : \widetilde{\texttt{Ref}}\,T \in \Gamma}{\Gamma \vdash x : \widetilde{\texttt{Ref}}\,T \rightsquigarrow x} \qquad \frac{\Gamma \vdash e : T \rightsquigarrow e'}{\Gamma \vdash \widetilde{\texttt{ref}}\,e : \widetilde{\texttt{Ref}}\,T \rightsquigarrow \texttt{ref}\,(e'\ \texttt{as}\ \star)}$$

$$\frac{\Gamma \vdash e : \widetilde{\texttt{Ref}}\,T \rightsquigarrow e'}{\Gamma \vdash !e : T \rightsquigarrow (!e')\ \texttt{as}\ T} \qquad \frac{\Gamma \vdash e_1 : \widetilde{\texttt{Ref}}\,T_1 \rightsquigarrow e'_1 \quad \Gamma \vdash e_2 : T_2 \rightsquigarrow e'_2 \quad T_1 \sim T_2}{\Gamma \vdash e_1 := e_2 : T_1 \rightsquigarrow (e'_1 := (e'_2\ \texttt{as}\ \star))\ \texttt{as}\ T_1}$$

Note that the static semantics for permissive references enforces type consistency at assignments, even though the assigned value is ultimately cast to $\star$. Furthermore, reference values translate to themselves, so object identity is preserved. However cast overhead is introduced at each dereference and update, so permissive references pay their own way with respect to performance.

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

$$\begin{array}{ll} (B \Rightarrow B) = \iota & (I \Rightarrow \star) = I! \\ (\star \Rightarrow \star) \;\; = \iota & (\star \Rightarrow I) = I? \end{array}$$

$$\begin{aligned} (T_1{\to}T_2) \Rightarrow (T_1'{\to}T_2') &= (T_1' \Rightarrow T_1){\to}(T_2 \Rightarrow T_2') \\ (T_1 \times T_2) \Rightarrow (T_1' \times T_2') &= (T_1 \Rightarrow T_1') \times (T_2 \Rightarrow T_2') \\ \mathtt{Ref}\,T \Rightarrow \mathtt{Ref}\,T' &= \mathtt{Ref}\,T' \end{aligned}$$

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

If we revisit the initial example in this section and replace `ref` with $\widetilde{\mathtt{ref}}$ and `Ref` with $\widetilde{\mathtt{Ref}}$, then this judgment translates the first program above into the second.

Proposition 1 (Translation). *If $\Gamma \vdash e : T \rightsquigarrow e'$ then $|\Gamma| \vdash e' : |T|$, Where $|\cdot|$ is the compatible extension of the equation $|\widetilde{\mathtt{Ref}}\,T| = \mathtt{Ref}\,\star$.*

This syntactic extension gives programmers access to both permissive references and monotonic references as desired.

Permissive references are a useful abstraction for the programmer and provide strong guarantees. However, such guarantees are provided only as long as permissive references do not flow into monotonic references. Consider the program above (with permissive references) where the following code comes after the `let` statements.

$$\begin{array}{l} \mathtt{let}\, w_1 = (x\ \mathtt{as}\ \star)\ \mathtt{in} \\ \mathtt{let}\, w_2 = (w_1\ \mathtt{as}\ \mathtt{Ref\,Bool})\ \mathtt{in} \\ w_2\ \mathtt{:=}\ \mathtt{true}; \end{array}$$

The program places us in a same situation as the original program that the monotonic semantics could not run without error. This example shows an important syntactic discipline for programmers that want to employ the monotonic paradigm for gradual references: *permissive references should not flow into monotonic references.*

4 Type Safety for Monotonic References

We present the high-points of the type safety proof here. The full proof is mechanized in Isabelle 2013 and available on arxiv [Siek and Vitousek, 2013]. 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 [Biernacka and Danvy, 2009].

We begin by lifting the precision relation to heap typings.

Definition 1 (Precision 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 more precise heap typing.

Lemma 1 (Strengthening wrt. the heap typing). *If $\Gamma; \Sigma \vdash e : T$ and $\Sigma' \sqsubseteq \Sigma$, then $\Gamma; \Sigma' \vdash e : T$.*

Proof (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 $\sqsubseteq$, we have $\Sigma'(a) \sqsubseteq T$. Therefore $\Gamma; \Sigma' \vdash a : T$.

The definition of well-typed heaps is standard.

Definition 2 (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 1 (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 1 we have $\emptyset; \Sigma' \vdash cv : T'$ and $\Sigma' \vdash \nu$. Thus, $\Sigma(a \mapsto T') \vdash \nu(a \mapsto cv : T')$.

Lemma 2 (Progress and Preservation). *Suppose $\emptyset; \Sigma \vdash e : T$ and $\Sigma \vdash \nu$. Exactly one of the following holds:*

1. *(a) e is a value, or*
   (b) $e =$ `error`, or
   (c) $e, \nu \longrightarrow e', \nu'$ for some e' and ν'.
2. *for all e', ν', if $e, \nu \longrightarrow e', \nu'$ then $\emptyset; \Sigma' \vdash e' : T$ and $\Sigma' \vdash \nu'$ and $\Sigma' \sqsubseteq \Sigma$ for some Σ'.*

Proof. We show a sketch of this proof in Appendix A.1.

Theorem 1 (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. *$e, \nu \longrightarrow^*$ `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 2 to conclude.

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

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

Fig. 9. Subtyping relation

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 Ref $\star$ to Ref Int and to Ref Bool.

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

With monotonic references, the cast at ℓ_3 triggers an error, because Int and 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 blame $\{\ell_2, \ell_3\}$.

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

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

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 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 Ref $\star$. The real culprit here is ℓ_1, which casts from Ref Int to Ref $\star$, thereby opening up the potential for the later cast error. Naïvely, this looks like an upcast, but a proper treatment of subtyping for references makes references invariant. So we have Ref Int $\not<:$ Ref $\star$ and the result of this program is blame $\{\ell_1\}$. Figure 9 presents the subtyping relation[3].

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

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 Boolean. Here is the first example:

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

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

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

The first example should produce $\texttt{blame}\,\{\ell_3\}$ while the second example should produce $\texttt{blame}\,\{\ell_4\}$, 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 as follows:

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

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

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

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

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.

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

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

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

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:

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

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

5.1 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 Figures 12 and 13. 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 2 (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 (A.2), 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

$$\begin{array}{lll} \text{Optional labels} & p, q & ::= \emptyset \mid \{\ell\} \\ \text{Label sets} & L & ::= \emptyset \mid \{\ell\} \mid \{\ell_1, \ell_2\} \\ \text{Labeled types} & P, Q & ::= B^p \mid P {\rightarrow}^p P \mid P {\times}^p P \mid \texttt{Ref}\,^p P \mid \star \end{array}$$

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

$$|B^p| = B \quad |P \rightarrow^p Q| = |P| \rightarrow |Q| \quad |P \times^p Q| = |P| \times |Q| \quad |\texttt{Ref}\,^p P| = \texttt{Ref}\,|P| \quad |\star| = \star$$

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

$$lab(B^p) = p \quad lab(P \rightarrow^p Q) = p \quad lab(P \times^p Q) = p \quad lab(\texttt{Ref}\,^p P) = p \quad lab(\star) = \emptyset$$

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

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

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

$$\begin{aligned} B^p \vartriangle B^q &= B^{p \vartriangle q} \\ P \vartriangle \star = P & \qquad \star \vartriangle Q = Q \\ (P \rightarrow^p P') \vartriangle (Q \rightarrow^q Q') &= (P \vartriangle Q) \hat{\rightarrow}^{p \vartriangle q} (P' \vartriangle Q') \\ (P \times^p P') \vartriangle (Q \times^q Q') &= (P \vartriangle Q) \hat{\times}^{p \vartriangle q} (P' \vartriangle Q') \\ \texttt{Ref}\,^p P \vartriangle \texttt{Ref}\,^q Q &= \hat{\texttt{Ref}}\,^{p \vartriangle q} (P \vartriangle Q) \\ P \vartriangle Q &= \bot^{lab(P) \cup lab(Q)} \qquad \text{otherwise} \end{aligned}$$

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

$$\begin{aligned} B^p \Leftrightarrow B^q &= B^{\emptyset} \\ P \Leftrightarrow \star = P & \qquad \star \Leftrightarrow Q = Q \\ (P \rightarrow^p P') \Leftrightarrow (Q \rightarrow^q Q') &= (P \Leftrightarrow Q) \hat{\rightarrow}^{\emptyset} (P' \Leftrightarrow Q') \\ (P \times^p P') \Leftrightarrow (Q \times^q Q') &= (P \Leftrightarrow Q) \hat{\times}^{\emptyset} (P' \Leftrightarrow Q') \\ \texttt{Ref}\,^p P \Leftrightarrow \texttt{Ref}\,^q Q &= \hat{\texttt{Ref}}\,^{\emptyset} (P \Leftrightarrow Q) \\ P \Leftrightarrow Q &= \bot^{lab(P) \cup lab(Q)} \qquad \text{otherwise} \end{aligned}$$

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

$$\begin{aligned} B^p \Rightarrow B^q = \iota & \qquad \star \Rightarrow \star = \iota \\ P \Rightarrow \star = P! & \qquad \star \Rightarrow Q = Q? \\ (P \rightarrow^p P') \Rightarrow (Q \rightarrow^q Q') &= (Q \Rightarrow P) \hat{\rightarrow} (P' \Rightarrow Q') \\ (P \times^p P') \Rightarrow (Q \times^q Q') &= (P \Rightarrow Q) \hat{\times} (P' \Rightarrow Q') \\ \texttt{Ref}\,^p P \Rightarrow \texttt{Ref}\,^q Q &= \hat{\texttt{Ref}}\,(P \Leftrightarrow Q) \\ P \Rightarrow Q &= \bot^{lab(P) \cup lab(Q)} \qquad \text{otherwise} \end{aligned}$$

Fig. 10. Labeled types and their operations

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

$$\begin{array}{llll} (B \Rightarrow^{\ell} B) = \iota & & (T \Rightarrow^{\ell} \star) = T^{\emptyset}! \\ (\star \Rightarrow^{\ell} \star) = \iota & & (\star \Rightarrow^{\ell} T) = T^{\ell}? \end{array}$$

$$(T_1 {\to} T_2) \Rightarrow^{\ell} (T_1' {\to} T_2') = (T_1' \Rightarrow^{\ell} T_1) {\to} (T_2 \Rightarrow^{\ell} T_2')$$

$$(T_1 \times T_2) \Rightarrow^{\ell} (T_1' \times T_2') = (T_1 \Rightarrow^{\ell} T_1') \times (T_2 \Rightarrow^{\ell} T_2')$$

$$\texttt{Ref}\, T_1 \Rightarrow^{\ell} \texttt{Ref}\, T_2 = \texttt{Ref}\, (T_1^{\ell} \Leftrightarrow T_2^{\ell})$$

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

$$B^{\ell} = B^{\ell} \quad (T_1 \to T_2)^{\ell} = T_1^{\ell} \to^{\ell} T_2^{\ell} \quad (T_1 \times T_2)^{\ell} = T_1^{\ell} \times^{\ell} T_2^{\ell}$$

$$(\texttt{Ref}\, T)^{\ell} = \texttt{Ref}\,^{\ell} T^{\ell} \quad \star^{\ell} = \star$$

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

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 $\triangle$, $\Rightarrow$, 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 $\widehat{\texttt{Ref}}$ 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 $\texttt{Ref Int}^{\ell}$ and T is $\texttt{Ref}\,\star$. Then the coercion we apply during the dereference is $\texttt{Int}^{\ell}!$; 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

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

Fig. 12. Syntax for monotonic references with blame

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

6 The Blame-Subtyping Theorem

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

We prove the blame-subtyping theorem via a preservation-style proof in which we preserve the $e\,\mathsf{safe}\,\ell$ predicate [Wadler and Findler, 2009]. This proof is 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 2. Recall that subtyping is defined in Figure 9 and compilation to coercions is defined in Figure 11. The safe predicate is defined for labeled type, coercions, expressions, and states in Figure 15.

Theorem 2 (Blame-Subtyping Theorem for the intermediate calculus). *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. We show a sketch of this proof in Appendix A.2.

Lemma 3 (Preservation of blame safety).
For all e, e', ν, ν', and ℓ, if $e, \nu\,\mathsf{safe}\,\ell$ and $e, \nu \longrightarrow e', \nu'$ then $e', \nu'\,\mathsf{safe}\,\ell$.

We now move away from the intermediate calculus and prove these important results on the gradually typed λ calculus with references. This latter language is indeed the one that programmers are expected to use. The following definitions will help to recast the results into the setting of the gradually typed language.

Definition 3 (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 4 (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.*

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

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

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

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

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

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

$$\frac{e \longrightarrow_c e'}{e, \nu \longrightarrow_{cr} e', \nu} \tag{P\textsc{Cast}B}$$

$$\frac{\begin{array}{c}\nu(a) = cv : P_1 \qquad P_3 = P_1 \vartriangle P_2 \\ |P_3| \neq |P_1| \quad cv' = cv\langle P_1 \Rightarrow P_3\rangle\end{array}}{a\langle \texttt{Ref}\, P_2\rangle, \nu \longrightarrow_{cr} a, \nu(a \mapsto cv' : P_3)} \tag{\textsc{Cast}R1B}$$

$$\frac{\nu(a) = cv : P_1 \qquad P_1 = P_1 \vartriangle P_2}{a\langle \texttt{Ref}\, P_2\rangle, \nu \longrightarrow_{cr} a, \nu} \tag{\textsc{Cast}R2B}$$

$$\frac{\nu(a) = cv : P_1 \qquad P_1 \vartriangle P_2 = \bot^L}{a\langle \texttt{Ref}\, P_2\rangle, \nu \longrightarrow_{cr} \texttt{blame}\, L, \nu} \tag{\textsc{Cast}R3B}$$

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

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

$$\texttt{!}\,a, \mu \longrightarrow_e \mu(a)_{\mathsf{val}}, \mu \tag{\textsc{Deref}MB}$$

$$\texttt{!}\,a@T, \mu \longrightarrow_e \mu(a)_{\mathsf{val}}\langle \mu(a)_{\mathsf{rtti}} \Rightarrow T^{\emptyset}\rangle, \mu \tag{\textsc{DynDrf}MB}$$

$$a \;\texttt{:=}\; v, \mu \longrightarrow_e a, \mu(a \mapsto v : \mu(a)_{\mathsf{rtti}}) \tag{\textsc{Upd}MB}$$

$$a \;\texttt{:=}\; v@T, \mu \longrightarrow_e a, \mu(a \mapsto cv : \mu(a)_{\mathsf{rtti}}) \tag{\textsc{DynUpd}MB}$$

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

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 $\boxed{e, \nu \longrightarrow e, \nu}$

$$\frac{e, \mu \longrightarrow_X e', \nu \qquad X \in \{cr, e\}}{e, \mu \longrightarrow e', \nu} \qquad \frac{\nu(a) = cv : P \qquad cv, \nu \longrightarrow_{cr} \texttt{blame}\, L, \nu'}{e, \nu \longrightarrow \texttt{blame}\, L, \nu'}$$

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

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

Fig. 13. Monotonic references with blame

We now have all the ingredients to state and prove one of the main contributions of the paper, i.e. the Blame-Subtyping Theorem for the gradually-typed λ calculus with references.

Lemma 4 (Translation preserves blame safety). *If* $e \,\mathsf{safe}\, \ell$ *and* $\Gamma \vdash e \leadsto e' : T$*, then* $e' \,\mathsf{safe}\, \ell$*.*

Proof. The proof is a straightforward induction on $\Gamma \vdash e \leadsto e' : T$.

Theorem 3 (Blame-Subtyping Theorem for the gradually-typed λ calculus with references).
For all e*,* e'*,* T_1*,* T_2*,* ℓ*, if* $\emptyset \vdash e \leadsto e' : T$*,* $e \,\mathsf{safe}\, \ell$*, and* $e', \emptyset \longrightarrow \mathtt{blame}\, L, \nu$*, then* $\ell \notin L$*.*

Proof. From the assumptions we have $e' \,\mathsf{safe}\, \ell$ by Lemma 4. Then we conclude by applying the Blame-Subtyping Theorem for the coercion calculus.

7 Implementation concerns w.r.t. 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 size, 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 pose a problem. This issue can be addressed using an approach inspired by garbage collection techniques. When the semantics is to update a cell with a larger value than the current one, the implementation allocates a new piece of memory and places a forwarding pointer in the old location. When reading and writing through dynamic references, the implementation must check for and follow the forwarding pointers. However, when reading and writing through fully-static references, the implementation does not need to consider forwarding pointers because fully-static heap cells never move. Then during a garbage collection, the implementation can collapse sequences of forwarding pointers to reduce overhead in subsequent execution.

8 Conclusion

We have presented a new design for gradually-typed mutable references, called monotonic references, the first to incur zero-overhead for reference accesses in statically typed code while maintaining the full expressiveness of a gradual type system. We defined a dynamic semantics for monotonic references and presented a mechanized proof of type safety. Further, we defined blame tracking based on using labeled types in the run-time type information and proved a blame theorem.

Bibliography

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

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

M. Biernacka and O. Danvy. Towards compatible and interderivable semantic specifications for the Scheme programming language, Part II. In *Semantics and Algebraic Specification*, pages 186–206, 2009.

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

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, ICFP, pages 48–59, 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. Microsoft Channel 9 Blog, April 2010.

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

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

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

A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & efficient gradual typing for TypeScript. Technical Report MSR-TR-2014-99, 2014.

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 M. M. Vitousek. Monotonic references for gradual typing. *Computing Research Repository*, 2013. URL http://arxiv.org/abs/1312.0694.

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, 2009.

T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: run-time support for reasonable interposition. OOPSLA, 2012.

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

S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium*, 2006.

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, 2010.

A Appendix

A.1 Type safety for monotonic references

Lemma 5 (Existence of meet for consistent types). *if $T_1 \sim T_2$ then there exists T_3 such that $T_3 = T_1 \sqcap T_2$.*

Lemma 6 (Meet is more precise). *if $T_3 = T_1 \sqcap T_2$ then $T_3 \sqsubseteq T_1$ and $T_3 \sqsubseteq T_2$.*

The proof of these two lemmas above are straightforward and omitted.

Lemma 2 (Progress and Preservation). *Suppose $\emptyset; \Sigma \vdash e : T$ and $\Sigma \vdash \nu$. Exactly one of the following holds:*

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

Proof (sketch).

Progress and Preservation for $\longrightarrow_{cr}$. As the definition of $\longrightarrow$ relies on $\longrightarrow_{cr}$, the proof of Lemma 2 requires that we prove progress and preservation for $\longrightarrow_{cr}$. We do not explicitly state this latter lemma, as it is identical to Lemma 2 except that is uses $\longrightarrow_{cr}$ in lieu of $\longrightarrow$. We first show a sketch of the proof of such a statement, this proof can be carried out with an induction on the derivation of $\emptyset; \Sigma \vdash e : T$. For $\longrightarrow_{cr}$, we show only one interesting case, namely when the provable derivation is $\emptyset; \Sigma \vdash a\langle \texttt{Ref}\, T\rangle : \texttt{Ref}\, T$.

Assume $\emptyset; \Sigma \vdash a\langle \texttt{Ref}\, T\rangle : \texttt{Ref}\, T$, and assume the proviso of the theorem. Since $\Sigma \vdash \nu$ we have that $\nu(a) = cv : T_1$ for some T_1. Now there are two cases, either $T \not\sim T_1$ or $T \sim T_1$, for both cases we show that a transition from $a\langle \texttt{Ref}\, T\rangle, \nu$ is provable. We first show the case when $T \not\sim T_1$, for which we can apply rule (CastRef3) as shown below.
Case $T \not\sim T_1$. We can instantiate rule (CastRef3) as follows.

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

Where the premises of this rule are satisfied by assumptions. Therefore $e, \nu \longrightarrow e', \nu'$ is provable for $e' = \texttt{error}$ and $\nu' = \nu$, that falls into the case 1.(c) of the progress property.

In the case $T \sim T_1$, Lemma 5 guarantees that $T_3 = T_1 \sqcap T$, for some T_3. We have now two cases, either $T_3 = T_1$ or $T_3 \neq T_1$. For both cases we show that a transition from $a\langle \texttt{Ref}\, T\rangle, \nu$ is provable. In particular by application of rule (CastRef1) and (CastRef2), respectively.

Case $T \sim T_1$ and $T_3 \neq T_1$. We can instantiate rule (CastRef1) as follows.

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

Where the premises of this rule are satisfied as described above, and $cv' = cv\langle T_1 \Rightarrow T_3 \rangle$ straightforwardly exists. Therefore $e, \nu \longrightarrow e', \nu'$ is provable for $e' = a$ and $\nu' = \nu(a \mapsto cv' : T_3)$, that falls into the case 1.(c) of the progress property.

Case $T \sim T_1$ and $T_3 = T_1$. We can instantiate rule (CastRef2) as follows.

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

Where the two premises are satisfied as described above. Therefore $e, \nu \longrightarrow e', \nu'$ is provable for $e' = a$ and $\nu' = \nu$, that falls into the case 1.(c) of the progress property.

Now we prove preservation for this case (derivation $\emptyset; \Sigma \vdash a\langle \mathtt{Ref}\, T\rangle : \mathtt{Ref}\, T$). We need to reason by case analysis on all the possible transitions from a $\langle \mathtt{Ref}\, T\rangle, \nu$. Those are only the ones provided by rules (CastRef1), (CastRef2) and (CastRef3) above. As we can reuse the instantiation above, we refer to them and do not repeat the rules. For case (CastRef1): We can pick $\Sigma' = \Sigma(a \mapsto T_3)$. Since $T_3 = T_1 \sqcap T$ we infer that $T_3 \sqsubseteq T$ by Lemma 6. Therefore $\Sigma' \sqsubseteq \Sigma$, and $\emptyset; \Sigma' \vdash a : \mathtt{Ref}\, T_3$ by Lemma 1. Moreover since $\Sigma' \sqsubseteq \Sigma$, $\nu(a \mapsto cv' : T_3) \sqsubseteq \nu$ follows by Corollary 1. For case (CastRef2): we can pick $\Sigma' = \Sigma(a \mapsto T_3)$ and apply the same reasoning as in the previous case for proving that in the step $a\langle \mathtt{Ref}\, T\rangle, \nu \longrightarrow_{cr} a, \nu$, the terms $a\langle \mathtt{Ref}\, T\rangle$ and a are typable with the same type. It is straightforward to see that also the rest holds. For case (CastRef3): We can pick $\Sigma' = \Sigma$. As `error` can be typed by any type, we have $\emptyset; \Sigma \vdash \mathtt{error} : \mathtt{Ref}\, T$. The rest trivially holds.

Progress and Preservation for $\longrightarrow$. We show only one interesting case, namely when $a \in \Sigma$ and $\nu(a)$ is not a value, i.e. $\nu(a) = cv : T_1$ for some cv and T_1 where cv is not a value. Since cv is not a value, we can apply progress of $\longrightarrow_{cr}$ and infer a step $cv, \nu \longrightarrow_{cr} cv', \nu_2$, for some cv' and ν_2. Now, we distinguish two cases depending on whether $\nu_2(a)_{\mathsf{rtti}} = T_1$ or $\nu_2(a)_{\mathsf{rtti}} \neq T_1$. For both cases, we can we prove that a transition from e, ν is provable. In particular by application of rule (HCast) and (HDrop), respectively.
Case $\nu_2(a)_{\mathsf{rtti}} = T_1$, we use (HCast): We can apply (HCast) instantiated as follows.

$$\frac{\begin{array}{c}\nu(a) = cv : T_1 \qquad cv, \nu \longrightarrow_{cr} cv', \nu_2 \\ \nu_2(a)_{\mathsf{rtti}} = T_1\end{array}}{e, \nu \longrightarrow e, \nu_2(a \mapsto cv' : T_1)}$$

Where all the premises are satisfied as described above. Therefore $e, \nu \longrightarrow e', \nu'$ is provable for $e' = e$ and $\nu' = \nu_2(a \mapsto cv' : T_1)$, that falls into the case 1.(c) of the progress property.
Case $\nu_2(a)_{\mathsf{rtti}} \neq T_1$, we use (HDrop): We can apply (HDrop) instantiated as follows.
Case (HDrop):

$$\frac{\nu(a) = cv : T_1 \qquad cv, \nu \longrightarrow_{cr} cv', \nu_2 \qquad \nu_2(a)_{\mathsf{rtti}} \neq T_1}{e, \nu \longrightarrow e, \nu_2}$$

Where all the premises are satisfied as described above. Therefore $e, \nu \longrightarrow e', \nu'$ is provable for $e' = e$ and $\nu' = \nu_2$, that falls into the case 1.(c) of the progress property.

We now proceed to prove preservation for this case. Let us consider the transitions given from the rules above. For (HCast): By type preservation of $\longrightarrow_{cr}$, there exists Σ_2 where $\Sigma_2 \vdash \nu_2$, $\emptyset; \Sigma_2 \vdash cv' : T_1$, and $\Sigma_2 \sqsubseteq \Sigma$. Take $\Sigma' = \Sigma_2$. Since $\emptyset; \Sigma \vdash e : T$ and $\Sigma' \sqsubseteq \Sigma$ we have $\emptyset; \Sigma' \vdash e : T$ by Lemma 1. Moreover, since $\nu_2(a)_{\mathsf{rtti}} = T_1$ we have that $\Sigma'(a) = T_1$ and since $\Sigma' \sqsubseteq \Sigma$ we can conclude $\Sigma' \vdash \nu_2(a \mapsto cv' : T_1)$ by Corollary 1.

The reader should notice that rule (HCast) is non-deterministic because there might be several instances of a for which the rule is satisfied. However, we have just proved that no matter whichever a the rule instantiation chooses, or to say, whichever a we are *challenged for*, we can construct the proof of preservation.

For (HDrop): By type preservation of $\longrightarrow_{cr}$, there exists Σ_2 where $\Sigma_2 \vdash \nu_2$ and $\Sigma_2 \sqsubseteq \Sigma$. Take $\Sigma' = \Sigma_2$. From $\emptyset; \Sigma \vdash e : T$ and $\Sigma' \sqsubseteq \Sigma$ we have $\emptyset; \Sigma' \vdash e : T$ by Lemma 1. Rule (HDrop), just as (HCast), is non-deterministic, however the same reasoning as above applies.

A.2 Proofs for the Blame-Subtyping Section (Section 6)

In this appendix, we first sketch the proof for Theorem 2 (Blame-Subtyping Theorem) and below the one for Lemma 3 (Preservation of blame safety). Necessary lemmas will be stated and proved.

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 7 (Reflexivity of $\Leftrightarrow$ and blame).
For all P and ℓ, $(P \Leftrightarrow P)$ safe ℓ.

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

Theorem 2 (Blame-Subtyping Theorem for the intermediate calculus). *For all T_1, T_2, and ℓ, it holds that $T_1 <: T_2$ iff $(T_1 \Rightarrow^{\ell} T_2)$ safe ℓ.*

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 `Ref` , as it relies on the operator $\Leftrightarrow$.

Case $\texttt{Ref}\,T_1' \Rightarrow^{\ell} \texttt{Ref}\,T_2' = \texttt{Ref}\,(T_1'^{\ell} \Leftrightarrow T_2'^{\ell})$: By definition of <:, we have $\texttt{Ref}\,T_1 <: \texttt{Ref}\,T_2$ only when $T_1 = T_2$. By Lemma 7 we have that $(T_1^{\ell} \Leftrightarrow T_1^{\ell})\,\mathsf{safe}\,\ell$, and thus $(\texttt{Ref}\,T_1' \Rightarrow^{\ell} \texttt{Ref}\,T_2')\ \mathsf{safe}\ \ell$.

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 $\rightarrow$ for it is contravariant.
Case $T_1 \rightarrow T_2 <: T_3 \rightarrow T_4$: To prove this, we need to derive $(T_3 <: T_1)$ and $(T_2 <: T_4)$. By the induction hypothesis we know that $(T_1 \rightarrow T_2) \Rightarrow^{\ell} (T_3 \rightarrow T_4)\ \mathsf{safe}\,\ell$. By definition, this means that $(T_3 \Rightarrow^{\ell} T_1) \rightarrow (T_2 \Rightarrow^{\ell} T_4)\ \mathsf{safe}\,\ell$. And therefore $(T_3 \Rightarrow^{\ell} T_1)\ \mathsf{safe}\ \ell$ and $(T_2 \Rightarrow^{\ell} T_4)\ \mathsf{safe}\,\ell$. 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 8 (Blame safety for $\vartriangle$, $\Rightarrow$ and $\Leftrightarrow$). *For all P and Q, if $P\,\mathsf{safe}\,\ell$ and $Q\ \mathsf{safe}\ \ell$ then $P \oplus Q\ \mathsf{safe}\,\ell$, 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 $\texttt{Ref}$ 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 3 (Preservation of blame safety).
For all e, e', ν, ν', and ℓ, if $e, \nu\ \mathsf{safe}\ \ell$ and $e, \nu \longrightarrow e', \nu'$ then $e', \nu'\ \mathsf{safe}\ \ell$.

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):

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

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 $P_2\ \mathsf{safe}\ \ell$. Because $P_3 = P_1 \vartriangle P_2$, by Lemma 8 (Blame Safety for $\vartriangle$) we infer $P_3\,\mathsf{safe}\,\ell$. Now, by Lemma 8 (Blame Safety for $\Rightarrow$), we have $P_1 {\Rightarrow} P_3\ \mathsf{safe}\,\ell$. Therefore $cv' = cv\langle P_1 {\Rightarrow} P_3\rangle\ \mathsf{safe}\,\ell$. Now, it is easy to see that $a, \nu(a \mapsto cv' : P_3)\,\mathsf{safe}\,\ell$, as $\nu\,\mathsf{safe}\,\ell$ by assumption and both $cv'\ \mathsf{safe}\ \ell$ and $P_3\ \mathsf{safe}\ \ell$ as inferred above.

(DynUpdMB):

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

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 $\mu(a)_{\mathsf{rtti}}\ \mathsf{safe}\,\ell$. Also, we can apply Lemma 8 (Blame Safety for $\Rightarrow$) to get $T^{\emptyset} \Rightarrow \mu(a)_{\mathsf{rtti}}\ \mathsf{safe}\,\ell$. Therefore $cv = v\langle T^{\emptyset} \Rightarrow \mu(a)_{\mathsf{rtti}}\rangle\ \mathsf{safe}\,\ell$. We finally conclude that $a, \mu(a \mapsto cv : \mu(a)_{\mathsf{rtti}})\ \mathsf{safe}\,\ell$.

B Auxiliary Definitions

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

$$\frac{}{\mathit{static}\ B} \qquad \frac{\mathit{static}\ T_1 \quad \mathit{static}\ T_2}{\mathit{static}\ T_1 \to T_2} \qquad \frac{\mathit{static}\ T_1 \quad \mathit{static}\ T_2}{\mathit{static}\ T_1 \times T_2} \qquad \frac{\mathit{static}\ T}{\mathit{static}\ \mathtt{Ref}\ T}$$

Fig. 14. Static types.

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

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

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

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

$$\frac{}{\iota\ \mathsf{safe}\ \ell} \qquad \frac{P\ \mathsf{safe}\ \ell}{P?\ \mathsf{safe}\ \ell} \qquad \frac{P\ \mathsf{safe}\ \ell}{P!\ \mathsf{safe}\ \ell} \qquad \frac{c_1\ \mathsf{safe}\ \ell \quad c_2\ \mathsf{safe}\ \ell}{c_1 \to c_2\ \mathsf{safe}\ \ell}$$

$$\frac{c_1\ \mathsf{safe}\ \ell \quad c_2\ \mathsf{safe}\ \ell}{c_1 \times c_2\ \mathsf{safe}\ \ell} \qquad \frac{c_1\ \mathsf{safe}\ \ell \quad c_2\ \mathsf{safe}\ \ell}{c_1\ ;\ c_2\ \mathsf{safe}\ \ell} \qquad \frac{P\ \mathsf{safe}\ \ell}{\mathtt{Ref}\ P\ \mathsf{safe}\ \ell}$$

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

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

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

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

Fig. 15. Definition of the safety predicate