Space-Efficient Monotonic References

DEYAAELDEEN ALMAHALLAWI, Indiana University Bloomington, USA
JEREMY G. SIEK, Indiana University Bloomington, USA

Integrating static typing and dynamic typing has received considerable attention in academia and industry. Gradual typing is one approach to this integration that preserves type soundness by casting values at run-time. For higher order values such as functions and mutable references, a cast typically wraps the values in a proxy that performs casts when the values are used. This approach suffers from two problems: (1) chains of proxies can grow and consume unbounded space, and (2) statically typed code regions need to check whether values are proxied. Monotonic references [Siek et al. 2015c] solve both problems for mutable references by directly casting the heap cell instead of wrapping the reference in a proxy. In this paper we show how to integrate monotonic references with the coercion-based solution to the first problem for other values such as functions [Herman et al. 2007]. We present a space-efficient version of the semantics for monotonic references, functions, and pairs and prove an upper bound on space overhead. Furthermore, the prior semantics for monotonic references involved storing cast expressions (not yet values) on the heap and it is not obvious how to implement this behavior efficiently in a compiler and run-time system. In our new semantics, only values are written to the heap, making the semantics straightforward to implement. Finally, we provide a type safety proof that is fully mechanized in Agda.

CCS Concepts: • **Software and its engineering → General programming languages**; **Semantics**.

Additional Key Words and Phrases: gradual-typing, reduction semantics, mutable state

ACM Reference Format:
Deyaaeldeen Almahallawi and Jeremy G. Siek. 2020. Space-Efficient Monotonic References. In *Informal Proceedings of the first ACM SIGPLAN Workshop on Gradual Typing (WGT20).* ACM, New York, NY, USA, 22 pages.

1 WHAT ARE THE PERFORMANCE CHARACTERISTICS OF GRADUAL TYPING?

Gradual typing combines static and dynamic type checking, giving the programmer control over which typing discipline to be used in each region of code [Anderson and Drossopoulou 2003; Gronski et al. 2006; Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006]. Siek et al. [2015b] describe five criteria for gradually typed languages, including *type soundness* and the *gradual guarantee*. The type soundness criteria requires that the value of an expression must have the type that was predicted by the type system. The gradual guarantee states that changing type annotations should not change the semantics of the program, except that incorrect type annotations may induce compile-time or run-time errors.

Type soundness is enforced, without compromising the expressive power of the language, by performing runtime type checking at the boundaries between statically typed and dynamically typed regions of code. This runtime overhead could be significant, depending on the nature of the language and what kind of features are supported. For instance, for languages with nominal types, the runtime check for whether a value has a given type is efficient and straightforward to implement. Indeed, Muehlboeck and Tate [2017] show that Nom, a nominally-typed object-oriented language (without generics or function types), exhibits low overheads. On the other hand, with structural types, the runtime check can be much more complex, e.g., for higher-order values it may involve the use of a proxy to mediate between a value and its context. For instance, casting

WGT20, January 25, 2020, New Orleans
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.

a function value wraps it in a proxy [Findler and Felleisen 2002] that checks, at application sites, whether the types of the input and output values matches that of the cast. Similarly, casting a reference value wraps the reference in a proxy that checks at read sites if the value on the heap could be cast to the type the context expects, and checks at write sites if the written value could be cast to the type of the heap cell.

Using proxies to cast higher-order values causes two problems. First, each time a value of a higher-order type crosses the boundary between a statically typed and dynamically typed code regions, it gets wrapped in a proxy. These chains of proxies can grow without bounds, causing space leaks Herman et al. [2007, 2010]. Furthermore, the chains of proxies can cause runtime slowdowns to the extent of changing the asymptotic complexity of a program. Kuhlenschmidt et al. [2019] demonstrate this problem with a simple quicksort program that is fully statically typed except for one annotation, causing a proxy to be created around the vector being sorted at each recursive call to the sort function. The runtimes collected from running the program on different input sizes suggest that the asymptotic complexity changes from $O(n^2)$ to $O(n^3)$.

The second problem with using proxies is the overhead of runtime dispatch on higher-order values at use sites. At each use site, the implementation must check whether the incoming value is proxied, and if so, perform runtime checks before and/or after processing the underlying value. This runtime dispatch is necessary even in statically typed code regions, because a reference from a dynamically typed region may flow into a static region. Kuhlenschmidt et al. [2019] observe about 20% overhead in a statically-typed matrix multiplication benchmark. This problem prevents the performance of statically-typed code in a gradually-typed programming language to be on par with corresponding code in statically-typed programming languages.

The good news is that these two problems have been solved for the case of functions. Herman et al. [2007, 2010] use the coercions of Henglein [1994] to compress chains by reducing sequences of coercions to normal form and Siek et al. [2015a] define a recursive function for composing coercions in normal form. Furthermore, Siek and Garcia [2012] solve the dynamic dispatch problem for functions, while also compressing coercions, by using a closure representation that can act as either a regular closure or as a proxy but with a uniform calling convention. This technique is used in Grift [Kuhlenschmidt et al. 2019].

For mutable references, Herman et al. [2007, 2010] solve the first problem by compressing chains of proxies using coercions, but their design does not solve the second problem of needing runtime dispatch to handle proxies. Siek et al. [2015c] introduce an alternative design, called *monotonic references*, that does not wrap addresses in proxies and instead casts values in the heap directly, effectively solving the dynamic dispatch issue. However, Siek et al. [2015c] leave as future work the challenge of identifying a normal form for coercions with monotonic references and of developing a function for composing such coercions. Furthermore, their monotonic heap maps addresses to *expressions* that are reduced using a small-step reduction relation. (Normally only values are stored in the heap.) It is not obvious how to implement this reduction-in-the-heap in a compiled implementation of a gradually-typed language.

In this paper, we answer the following questions.

Q.1 How to integrate monotonic references into a space-efficient coercion calculus?
Q.2 How to implement monotonic references in a way that only puts values in the heap?

We answer those questions by presenting a new reduction semantics for monotonic references that is space-efficient for functions and that writes values only to the heap. To achieve the former, we interpret casts as coercions [Henglein 1994; Herman et al. 2007, 2010] and extend the coercion composition operator from prior work [Siek et al. 2015a], to also compose coercions for monotonic references. Regarding the latter, we provide a cast function that returns a value (that can then

be written to the heap) and a list of casts on addresses that are put into a queue for subsequent processing, making the semantics straightforward to implement.

The rest of the paper is organized as follows: Section 2 illustrates the problem of chains of function proxies and the importance of space-efficiency. Section 3 illustrates why prior work on monotonic references wrote cast expressions to the heap. Section 4 gives an informal presentation of our approach. Sections 5 and 6 present our approach formally and prove type safety.

2 THE PROBLEM OF CHAINS OF FUNCTION PROXIES

Proxies ensure that the type of a proxied value matches a certain type. Proxies are needed when it is hard to check immediately if the type of a value matches some type. For instance, it is in general impossible to check whether an arbitrary untyped function will return a Boolean without applying it first. With proxies, the function gets wrapped in a proxy that performs the desired checks at application sites. In this case, it checks if the incoming argument can be cast to the domain type of the underlying function, and also checks if the return value could be cast to Boolean.

An important consequence of that wrapping, however, is that values could accumulate multiple proxies when flowing through the boundaries between statically typed and dynamically typed regions of code. Operations on such proxied values have to go through all the layers of proxies, performing all their checks in order to access the underlying value.

This problem was first observed by Herman et al. [2007, 2010] in two higher-order, mutually recursive functions. One of the functions expects a statically typed function as input and the other expects a dynamically typed function, so implicit function casts are inserted to mediate between the types. A function cast creates a run-time proxy wrapping the function being cast. Every time one of the functions is called, a new proxy is created. These proxies consume space proportional to the number of recursive calls.

With monotonic references there are no proxies on references, however the prior work did not integrate the solutions for space efficiency on other higher-order values such as functions [Siek et al. 2015c]. In this paper we combine monotonic references with space-efficiency for other values.

3 THE PROBLEM OF EXPRESSIONS ON THE HEAP

In this section we review how and why the prior semantics for monotonic references puts expressions on the heap [Siek and Vitousek 2013; Siek et al. 2015c]. Consider the following example that creates a cycle in the heap, written in a variant of Typed Racket syntax that supports the dynamic type `Dyn` to enable fine-grained gradual typing. (Typed Racket uses the term "box" to mean reference. The `Boxof` type corresponds to the Ref type.)

```
1 (: f Dyn)
2 (define f (lambda (x) x))
3 (: r Dyn)
4 (define r (box (cons f '())))
5 (set-box! r (cons f r)) ;; establish a cycle
6 (define (g [x : (Boxof (Pair (Dyn -> Integer) (Boxof (Pair (Integer -> Dyn) Dyn))))])
7   ((car (unbox x)) 42))
8 (g r)
```

On line 2, a reference is allocated at some address, call it a, and placed in variable r; the heap cell at a is initialized with a pair of values of dynamic type. On line 3, a cycle is created by assigning to r another pair whose second element is r. So at this point the heap is in the following state.

$$\{a \mapsto \langle(\lambda x.x)\langle\star \to \star \Rightarrow \star\rangle, a\langle\text{Ref } (\star \times \star) \Rightarrow \star\rangle\rangle\}$$

On line 6, function g is applied to r, but g expects a reference to a pair of type

$$(\star \to \mathsf{Int}) \times \mathsf{Ref}\ ((\mathsf{Int} \to \star) \times \star)$$

So a cast will be performed before proceeding with the function application. Casting the pair updates address a with the following expression, where values are colored purple and casts that are about to be applied are colored magenta:

$$a \mapsto \left\langle \begin{array}{l} (\lambda x.x)\langle \star \to \star \Rightarrow \star \rangle \langle \star \Rightarrow (\star \to \mathsf{Int}) \rangle, \\ a\langle \mathsf{Ref}\ (\star \times \star) \Rightarrow \star \rangle \langle \star \Rightarrow \mathsf{Ref}\ ((\mathsf{Int} \to \star) \times \star) \rangle \end{array} \right\rangle$$

The cast on the first element of the pair proceeds without further ado.

$$a \mapsto \left\langle \begin{array}{l} (\lambda x.x)\langle \star \to \star \Rightarrow \star \to \mathsf{Int} \rangle, \\ a\langle \mathsf{Ref}\ (\star \times \star) \Rightarrow \star \rangle \langle \star \Rightarrow \mathsf{Ref}\ ((\mathsf{Int} \to \star) \times \star) \rangle \end{array} \right\rangle$$

The cast on the second element of the pair is a cast to a reference type, so it will cast address a to the greatest lower bound (with respect to type precision) of its current type and the target type of the cast, which amounts to:

$$(\mathsf{Int} \to \mathsf{Int}) \times \mathsf{Ref}\ ((\mathsf{Int} \to \star) \times \star)$$

This cast is interesting because it causes another cast on the first element of the pair. Proceeding with a reduction step yields the following state.

$$a \mapsto \left\langle \begin{array}{l} (\lambda x.x)\langle \star \to \star \Rightarrow \star \to \mathsf{Int} \rangle \langle \star \to \mathsf{Int} \Rightarrow \mathsf{Int} \to \mathsf{Int} \rangle, \\ a \end{array} \right\rangle$$

In the above sequence of reductions, it was crucial that the cast on the first element of the pair to $\star \to \mathsf{Int}$ was written to the heap before the first element was cast again to the type $\mathsf{Int} \to \star$, enabling a greatest lower bound of $\mathsf{Int} \to \mathsf{Int}$ that took both types into account.

Unfortunately, it is not obvious how to implement this process efficiently in a compiler and runtime system. Literally placing expressions in the heap would require a runtime representation of abstract syntax trees and an interpreter, which would be costly to implement and rather slow to execute.

Alternatively, one might try to cast the value in a heap cell in one big step before writing it back to the heap, but the straightforward version of this idea is problematic because of race conditions. In the above example, the second cast on the first element, to $\mathsf{Int} \to \star$, would see the current type as still $\star \to \star$, so the greatest lower bound would be $\mathsf{Int} \to \star$. This would break the monotonicity property (and therefore type safety), because $\mathsf{Int} \to \star$ is not less precise than $\star \to \mathsf{Int}$. In this paper we present a refinement of the big-step approach. The key insight is that in the process of casting the value in a heap cell, when we come to an address, instead of immediately casting it, we leave the address as-is and instead put the address and the target type of the cast in a queue. Once the entire value has been processed, we write the result value back to the heap and then cast the addresses in the queue.

4 SPACE-EFFICIENCY AND VALUES ON THE HEAP

This section describes our approach to solve the two problems explained in Sections 2 and 3. Section 4.1 describes the design of normal coercions for monotonic references and Section 4.2 describes our design of the new reduction semantics for monotonic references that only writes values to the heap.

4.1 Coercions in Normal Form for Monotonic References

To answer **Q.1**, we extend the prior work on coercions by [Kuhlenschmidt et al. 2019], designing normal coercions for monotonic references and extending the coercion composition operator to handle coercions that involve monotonic references.

A monotonic reference coercion holds just a type. To cast an address with some reference coercion Ref T, we cast the value at that address from its type to the greatest lower bound of its type and T. For convenient access, the type of a heap cell is stored next to the cell in the heap, and we refer to it as the run-time type information (RTTI). Casting an address with a sequence of two reference coercions amounts to taking the greatest lower bound of the RTTI and the types those two coercions hold. This suggest that composing two reference coercions should create a new reference coercion that holds a type that is the result of taking the greatest lower bound of the types of the coercions being composed. However this definition causes subtle changes to the coercion grammar and typing. This will be explained in more detail in Section 5.

4.2 Queue of Suspended Casts

To answer **Q.2**, we developed a new dynamic semantics for monotonic references, named λ_S^{ref}, that does not involve storing expressions on the heap. Instead, it uses a simple recursive function to cast values in the heap down to values. To maintain the correctness of reference casts in the presence of cycles, the function does not apply any inner reference casts. Instead it returns the address and puts the inner reference cast in a queue. Reduction proceeds by examining the queue, so a suspended cast is picked from the queue and gets applied. Applying a cast can return more suspended casts, so those get appended to the queue. This process continues until the queue is empty.

The main insight behind this design is that the correct ordering of cast reduction is a top-down breadth first traversal, as discussed in Section 3. With this semantics, the example in Section 3 can be reduced correctly. Recall that the value at address a is allocated at type $\star \times \star$ and is cast to

$$(\star \to \mathsf{Int}) \times \mathsf{Ref}\ ((\mathsf{Int} \to \star) \times \star)$$

Because the top-level cast is a reference cast, it will be pushed to the queue. Subsequently, the cast will be read from the queue and applied to the pair in the heap. The cast on the first projection will be $\star \Rightarrow \star \to \mathsf{Int}$ and will fully reduce to a value. The cast on the second projection will be suspended by reducing to the address a and the cast itself will be pushed to the queue. So this value will be written to the heap as follows. (The RTTI for the heap cell is written after the colon.)

$$\{a \mapsto \langle \lambda x.x \langle \star \to \star \Rightarrow \star \to \mathsf{Int} \rangle, a \rangle : (\star \to \mathsf{Int}) \times \mathsf{Ref}\ ((\mathsf{Int} \to \star) \times \star)\}$$

and the queue will contain the address a with the target type $(\mathsf{Int} \to \star) \times \star$. Subsequently, the address and its target type will be popped from the queue. The value at the address is cast from its RTTI to the greatest lower bound of the RTTI and the target type. This results in the following final state:

$$\{a \mapsto \langle \lambda x.x \langle \star \to \star \Rightarrow \mathsf{Int} \to \mathsf{Int} \rangle, a \rangle : (\mathsf{Int} \to \mathsf{Int}) \times \mathsf{Ref}\ ((\mathsf{Int} \to \star) \times \star)\}$$

Note that casting the function for the second time composed the old cast $\star \to \star \Rightarrow \star \to \mathsf{Int}$ and the new one $\star \to \mathsf{Int} \Rightarrow \mathsf{Int} \to \mathsf{Int}$ into one cast $\star \to \star \Rightarrow \mathsf{Int} \to \mathsf{Int}$. This is a result of our semantics being space-efficient. (The details of this composition is omitted here to simplify the presentation of the reduction.)

It is worth noting that the length of the queue is bounded by the size of the value being cast. This means that the queue cannot grow without bound, so the space overhead caused by the queue is always a constant factor. This new approach is easier to understand and is straightforward to implement in a runtime system for gradually-typed languages.

5 COERCIONS IN NORMAL FORM FOR MONOTONIC REFERENCES

Coercions are combinators on types, originally designed for compile-time optimization for dynamically-typed languages [Henglein 1994]. They were later use by Herman et al. [2007, 2010] to guarantee space-efficiency at run-time for gradual typing. We follow suit and use coercions to guarantee space-efficiency in λ_S^{ref}, answering question **Q.1**.

There are many semantics for coercions [Siek et al. 2009]. In this paper, we use the Lazy-D semantics which allows any type, other than $\star$ itself, to be injected directly to $\star$. Let I and J range over types other than $\star$. We adapt the definition of coercions for monotonic references from Siek et al. [2015c], adding a failure coercion of the form $\bot^{I,J}$.

$$c, d ::= \iota \mid I! \mid I? \mid c \to d \mid c \times d \mid \text{Ref } T \mid \bot^{I,J}$$

To review, the identity coercion ι acts as the identity function. The coercion $I!$ is an injection into $\star$ that tags v with the source type. The projection coercion $I?$ projects the tagged value v from $\star$ to I by checking if the tag is consistent with I. If this is the case, the underlying value is returned. The failure coercion $\bot$ signals an error when the source and target types are not consistent. The function coercion $c \to d$ creates a proxy around the function v that applies coercion c to the argument and coercion d to the return value. The product coercion $c \times d$ applies coercion c to the first projection and coercion d to the second projection. Finally, and most relevant to this paper, the reference coercion Ref T casts the value in the heap to be at least as precise as T.

We now turn to identifying a normal form for coercions with monotonic references and a function for composing them. We adapt the coercions in normal form of Kuhlenschmidt et al. [2019] (see the appendix in the auxiliary archive). Coercions in normal form are defined by a grammar consisting of three rules that enable coercion composition by a straightforward recursive function. This restricted grammar makes sure that the longest coercion will be one that starts with a projection, followed by a middle coercion, and ends with an injection. We adapt the normal form of Kuhlenschmidt et al. [2019] by replacing the traditional reference coercion with one for monotonic references, removing the duplicate identity coercion ι, and moving the failure coercion from final coercions to middle coercions. Figure 1 presents our coercions in normal form and how to create and compose them, which we describe next.

Our composition function $\fatsemi$ takes two coercions in normal form as input and returns a coercion in normal form. It relies on two helper functions: $\fatsemi_i$ that composes a final coercion with a coercion in normal form and returns a coercion in normal form, and $\fatsemi_g$ that composes two middle coercions and returns a middle coercion.

Composing reference coercions. Composing two traditional reference coercions is done by composing the coercions for the read and the coercions for the write [Herman et al. 2007, 2010]. Composing two monotonic coercions is different because they hold types instead that are used to cast the heap. We merge the type of the two coercions by taking the greatest lower bound with respect to the precision relation $\sqsubseteq$ (using the function $\sqcap$). However, the function $\sqcap$ is partial and is undefined when the two input types are not consistent. In that case we return the failure coercion $\bot$ instead, which is why we move failure coercions into the grammar for middle coercions.

Our coercion composition function maps two well-typed coercions to a well-typed coercion.

Lemma 5.1 (Well-typed Composition). *if* $c : T_1 \Longrightarrow T_2$ *and* $d : T_2 \Longrightarrow T_3$, *then* $c \fatsemi d : T_1 \Longrightarrow T_3$

Proof. For the case Ref $T \fatsemi_g$ Ref T': We know that $T' \sqsubseteq T_3$ by Ref $T' : T_2 \Longrightarrow T_3$. We also know that $(T \sqcap T') \sqsubseteq T'$. By this and the transitivity of $\sqsubseteq$, we conclude that $(T \sqcap T') \sqsubseteq T_3$. □

Our proof of that composition terminates relies on the fact that the sum of the sizes of the input coercions gets smaller at each recursive call.

Proposition 5.2 (Termination of Composition). $size(c \fatsemi d) \leq size(c) + size(d)$

To establish space-efficiency, we reason about the height of the created and composed coercions. The definition of height we use is different from the one of Herman et al. [2007] because of the difference in semantics between Lazy-UD and Lazy-D. Instead of injection and projection coercions having a height of 1 in the former, they instead have the height of the types they carry.

Lemma 5.3 (Coercions height is bounded). $\|T_1 \Rightarrow T_2\| \leq \max(\|T_1\|, \|T_2\|)$

Proof. By induction on the coercion creation $T_1 \Rightarrow T_2$. □

Proposition 5.4 (Composition height is bounded). $\|c \fatsemi d\| \leq \max(\|c\|, \|d\|)$

Proof. By Lemma 5.3 and induction on the structure of the compose function $c \fatsemi d$. □

A well-typed coercion in normal form consists of at most two sequences (three coercions or less), so a space-efficient coercion bounded in height is also bounded in size.

6 λ_S^{ref} DYNAMIC SEMANTICS

In this section, we present the reduction semantics for λ_S^{ref}, and show how it answers both questions **Q.1** and **Q.2**. Most of our answer to question **Q.1** has been presented in Section 5. What remains to show is how to restrict the reduction relation to use the composition function $\fatsemi$ to combine adjacent coercions first before reducing expressions underneath. The approach we use is fairly standard [Herman et al. 2007, 2010; Siek et al. 2015a; Siek and Wadler 2010]. We adjust our notion of values, defined in Figure 2, to only allow at most one layer of casts. Then, a new rule for composing coercions is introduced in the pure coercion reduction relation $\longrightarrow_c^{\fatsemi}$, defined in Figure 3. Finally, expression reduction relation $\longrightarrow_e$, defined in Figure 4, is restricted such that reduction is disallowed under a sequence of two or more casts. Typically, this restriction is implemented by introducing a notion of cast-free evaluation context that does not have a cast application innermost and requiring the reduction of cast expressions to occur in that cast-free context. Instead, we use frames, which is a much simpler structure than evaluation contexts, and there is no need to worry about different ways to decompose them. The frame for cast expressions is removed from the grammar for frames, defined in Figure 2, and congruence rules for cast expressions is added to $\longrightarrow_e$. Furthermore, the reduction relation is indexed by a flag that indicates whether the cast congruence rule is allowed (cca) in the current context or whether it is disallowed (ccd). The cast congruence rules, CastCong and CastCongE, require the flag cca, and they switch the flag of the subexpression reduction to ccd. The rest of the rules are indexed by ccd. To allow rules indexed by ccd to be also available in cca contexts, we include a subsumption rule, Switch, that flips the flag from cca to ccd. This completes the presentation of our setup for space efficiency.

On the other hand, answering question **Q.2** requires the definition of the heap to be a map from addresses to pairs of values and types (Figure 2). This is achieved by changing how reference reduction works as follows. Instead of writing a delayed cast expression to the heap, reducing a reference cast creates a suspended cast and put it in a queue on the side (referred to by Ψ), without actually changing the heap. This is done using the rule MonoCast of the program reduction relation $\longrightarrow_e$, defined in Figure 4. The state reduction relation $\longrightarrow$, defined in Figure 4, pauses expression reduction if the queue is not empty and pops a suspended cast to reduce. For the suspended cast on address a, the value mapped to this address gets cast from $\mu(a)_{\text{rtti}}$ to $\mu(a)_{\text{rtti}} \sqcap T$ using the apply-cast function, defined in Figure 3. The latter function fully reduces the cast to a value while also suspending any inner reference casts creating another queue of suspended casts. The result value gets written to the heap and the new queue gets appended to the old one. This

$$
\begin{array}{lllll}
\text{Coercions (in normal form)} & c, d & ::= & (I?\,;\, i) \mid i \\
\text{Final coercions} & i & ::= & (g\,;\, I!) \mid g \\
\text{Middle coercions} & g & ::= & \iota \mid c \to d \mid c \times d \mid \mathsf{Ref}\ T \mid \bot
\end{array}
$$

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

$$
\frac{}{\iota : T \Longrightarrow T} \qquad \frac{g : T \Longrightarrow I}{g\,;\, I! : T \Longrightarrow \star} \qquad \frac{i : I \Longrightarrow T}{I?\,;\, i : \star \Longrightarrow T} \qquad \frac{}{\bot : T \Longrightarrow T'}
$$

$$
\frac{c : T_1' \Longrightarrow T_1 \qquad d : T_2 \Longrightarrow T_2'}{c \to d : T_1 \to T_2 \Longrightarrow T_1' \to T_2'} \qquad \frac{c : T_1 \Longrightarrow T_1' \qquad d : T_2 \Longrightarrow T_2'}{c \times d : T_1 \times T_2 \Longrightarrow T_1' \times T_2'} \qquad \frac{T \sqsubseteq T_2}{\mathsf{Ref}\ T : \mathsf{Ref}\ T_1 \Longrightarrow \mathsf{Ref}\ T_2}
$$

Middle coercion creation $\boxed{(I \Rightarrow_g I) = i}$

$$
\begin{array}{l}
(I \Rightarrow_g I) = \iota \\
(I \Rightarrow_g J) = \bot \qquad \text{if } I \not\sim J
\end{array}
$$

$$
\begin{array}{r}
(T_1 \to T_2 \Rightarrow_g T_1' \to T_2') = (T_1' \Rightarrow T_1) \to (T_2 \Rightarrow T_2') \\
(T_1 \times T_2 \Rightarrow_g T_1' \times T_2') = (T_1 \Rightarrow T_1') \times (T_2 \Rightarrow T_2') \\
(\mathsf{Ref}\ T \Rightarrow_g \mathsf{Ref}\ T') = \mathsf{Ref}\ T'
\end{array}
$$

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

$$
\begin{array}{rclrcl}
(I \Rightarrow \star) & = & \iota\,;\, I! & (\star \Rightarrow \star) & = & \iota \\
(\star \Rightarrow I) & = & I?\,;\, \iota & (I \Rightarrow J) & = & I \Rightarrow_g J
\end{array}
$$

Middle coercion composition $\boxed{g \mathbin{⨟_g} g = g}$

$$
\begin{array}{rcl}
\bot \mathbin{⨟_g} g & = & \bot \\
g \mathbin{⨟_g} \bot & = & \bot \\
\iota \mathbin{⨟_g} g & = & g \\
g \mathbin{⨟_g} \iota & = & g \\
c \to d \mathbin{⨟_g} c' \to d' & = & (c' \mathbin{⨟} c) \to (d \mathbin{⨟} d') \\
c \times d \mathbin{⨟_g} c' \times d' & = & (c \mathbin{⨟} c') \times (d \mathbin{⨟} d') \\
\mathsf{Ref}\ T \mathbin{⨟_g} \mathsf{Ref}\ T' & = & \bot \quad \text{if } T \nsim T' \\
\mathsf{Ref}\ T \mathbin{⨟_g} \mathsf{Ref}\ T' & = & \mathsf{Ref}\ (T \sqcap T') \quad \text{if } T \sim T'
\end{array}
$$

Final coercion composition $\boxed{i \mathbin{⨟_i} c = c}$

$$
\begin{array}{rcl}
(g\,;\, I!) \mathbin{⨟_i} (J?\,;\, i) & = & g \mathbin{⨟_i} ((I \Rightarrow_g J) \mathbin{⨟_i} i) \\
g \mathbin{⨟_i} (g'\,;\, I!) & = & (g \mathbin{⨟_g} g')\,;\, I! \\
g \mathbin{⨟_i} g' & = & g \mathbin{⨟_g} g' \\
\bot \mathbin{⨟_i} i & = & \bot \\
i \mathbin{⨟_i} \bot & = & \bot \\
\iota \mathbin{⨟_i} i & = & i \\
i \mathbin{⨟_i} \iota & = & i
\end{array}
$$

Coercion composition $\boxed{c \mathbin{⨟} c = c}$

$$
(I?\,;\, i) \mathbin{⨟} c = I?\,;\, (i \mathbin{⨟_i} c) \qquad i \mathbin{⨟} c = i \mathbin{⨟_i} c
$$

Type height $\boxed{\|T\|}$

$$
\|\mathsf{Int}\| = 1 \qquad \|()\| = 1 \qquad \|\star\| = 1 \qquad \|T_1 \to T_2\| = 1 + max(\|T_1\|, \|T_2\|)
$$

$$
\|T_1 \times T_2\| = 1 + max(\|T_1\|, \|T_2\|) \qquad \|\mathsf{Ref}\ T\| = 1 + \|T\|
$$

Coercion height $\boxed{\|c\|}$

$$
\|\iota\| = 1 \qquad \|\bot\| = 1 \qquad \|\mathsf{Ref}\ T\| = 1 + \|T\| \qquad \|c \to d\| = 1 + max(\|c\|, \|d\|)
$$

$$
\|c \times d\| = 1 + max(\|c\|, \|d\|) \qquad \|c\,;\, I!\| = max(\|c\|, \|I\|) \qquad \|I?\,;\, c\| = max(\|I\|, \|c\|)\|
$$

Fig. 1. Normal coercions and their operations

$$
\begin{array}{llll}
\text{Uncoerced Values} & u & ::= & k \mid \lambda(x : T).\, e \mid \langle v, v \rangle \mid a \\
\text{Values} & v & ::= & u \mid u\langle c \to d \rangle \mid u\langle g\,;\, I! \rangle \\
\text{Heap} & \mu & ::= & \emptyset \mid \mu(a \mapsto v : T) \\
\text{SuspendedCasts} & \Psi & ::= & \emptyset \mid (a, T); \Psi \\
\text{Frames} & F & ::= & \Box\, e \mid v\, \Box \mid \langle \Box, e \rangle \mid \langle v, \Box \rangle \mid \mathtt{fst}\ \Box \mid \mathtt{snd}\ \Box \mid \\
 & & & \mathtt{ref}\ \Box @ T \mid\ !\Box @ T \mid\ !\Box \mid \Box := e @ T \mid v := \Box @ T
\end{array}
$$

Fig. 2. Run-time structures for the space-efficient coercion calculus

$$
\begin{aligned}
\text{apply-cast}((u\langle c \rangle), d) &= \text{apply-cast}(u, (c \fatsemi d)) \\
\text{apply-cast}(a, (\text{Ref } T)) &= \text{just } ((a, T); \emptyset, a) \\
\text{apply-cast}(u, \iota) &= \text{just } (\emptyset, u) \\
\text{apply-cast}(u, (c \to d)) &= \text{just } (\emptyset, u\langle c \to d \rangle) \\
\text{apply-cast}(u, (g\,;\, I!)) &= \text{just } (\emptyset, u\langle g\,;\, I! \rangle) \\
\text{apply-cast}(\langle v_1, v_2 \rangle, (c \times d)) &= \text{just } (\Psi \oplus \Psi', \langle v_1', v_2' \rangle) \\
&\qquad \text{if } \begin{array}{l} \text{apply-cast}(v_1, c) = \text{just } (\Psi, v_1'), \\ \text{apply-cast}(v_2, d) = \text{just } (\Psi', v_2') \end{array} \\
\text{apply-cast}(\_, \_) &= \text{nothing}
\end{aligned}
$$

Fig. 3. apply-cast applies a coercion to a value and possibly returns a value and a queue of suspended casts.

process continues until there is no more suspended casts in the queue. Now the queue is empty, the expression at hand can resume reduction. This way, only values are written to the heap.

State reduction. The state reduction relation $\longrightarrow$ contains 5 rules, 4 of them deal with casting the heap with a suspended cast from the queue. A step can be taken via rule UpdateHeap if the suspended cast (a, T) is productive, i.e. $T \sqcap \mu(a)_{\text{rtti}} \neq \mu(a)_{\text{rtti}}$, and successful, the result value of the call to apply-cast gets written to the heap and the RTTI is updated accordingly. On the other hand, if the cast fails, the Error1 rule reduces to error. Moreover, if the cast is not productive, i.e. $T \sqcap \mu(a)_{\text{rtti}} = \mu(a)_{\text{rtti}}$, the NoChange rule drops it from the queue. Finally, if the target type of the suspended cast is not consistent with the RTTI, the Error2 rule reduces to error.

Note that the apply-cast function maintains space-efficiency by composing the old coercion on the value being cast and the new coercion.

Example. Now let's consider how the example program in Section 3 reduces with the dynamic semantics of λ_S^{ref}. The cast expression we want to reduce is:

$$r\langle(\text{Ref } ((\star \to \text{Int}) \times \text{Ref } ((\text{Int} \to \text{Int}) \times \star)))\rangle$$

and the heap is:

$$a \mapsto \langle f\langle \iota\,;\, (\star \to \star)! \rangle, a\langle \iota\,;\, (\text{Ref } (\star \times \star))! \rangle\rangle : \star \times \star$$

Reduction will proceed by applying rule Cast/Succeed reducing the expression to the address a and putting the cast in the queue. The queue Ψ will be $(a, (\star \to \text{Int}) \times \text{Ref } ((\text{Int} \to \text{Int}) \times \star)), \emptyset$. Next, rule UpdateHeap will be used to pop the cast from the queue, apply it to the value in the heap, write the new value and the new RTTI to the heap, and add the inner reference cast to the queue. The heap will be updated as follows:

$$a \mapsto \langle f\langle(\iota \to (\text{Int?}\,;\, \iota))\rangle, a\rangle : (\star \to \text{Int}) \times \text{Ref } ((\text{Int} \to \text{Int}) \times \star)$$

and the queue will be $(a, (\text{Int} \to \text{Int}) \times \star), \emptyset$. Finally, reduction will apply UpdateHeap to perform the cast in the queue. The final heap will be:

Pure reduction rules $\boxed{e \longrightarrow^{\oslash}_{e} e}$

$$\begin{array}{llll} \texttt{fst}\ \langle v_1, v_2\rangle & \longrightarrow^{\oslash}_{e} & v_1 & (\lambda(x:T).\,M)\ v \longrightarrow^{\oslash}_{e} [x := v]M \\ \texttt{snd}\ \langle v_1, v_2\rangle & \longrightarrow^{\oslash}_{e} & v_2 & (u\langle c \to d\rangle)\ v \longrightarrow^{\oslash}_{e} (u\,(v\langle c\rangle))\langle d\rangle \end{array}$$

Reference operations reduction rules $\boxed{e, \mu \longrightarrow^{\bigcirc}_{e} e, \mu, \Psi}$

$$\texttt{ref}\ v@T, \mu \longrightarrow^{\bigcirc}_{e} a, \mu(a \mapsto v : T), \emptyset \quad \text{if } a \notin \text{dom}(\mu) \qquad \text{(Alloc)}$$

$$!a, \mu \longrightarrow^{\bigcirc}_{e} \mu(a)_{\text{val}}, \mu, \emptyset \qquad \text{(Read)}$$

$$!a@T, \mu \longrightarrow^{\bigcirc}_{e} \mu(a)_{\text{val}}\langle \mu(a)_{\text{rtti}} \Rightarrow T\rangle, \mu, \emptyset \qquad \text{(DynRead)}$$

$$a := v, \mu \longrightarrow^{\bigcirc}_{e} \text{Unit}, \mu(a \mapsto v : \mu(a)_{\text{rtti}}), \emptyset \qquad \text{(Update)}$$

$$a := v@T, \mu \longrightarrow^{\bigcirc}_{e} \text{Unit}, \mu(a \mapsto v' : \mu(a)_{\text{rtti}}), \Psi \quad \text{if apply-cast}(v, (T \Rightarrow \mu(a)_{\text{rtti}})) = \text{just}\ (\Psi, v') \qquad \text{(DynUpdate)}$$

$$a := v@T, \mu \longrightarrow^{\bigcirc}_{e} \texttt{error}, \mu, \emptyset \quad \text{if apply-cast}(v, (T \Rightarrow \mu(a)_{\text{rtti}})) = \text{nothing} \qquad \text{(DynUpdateFail)}$$

Program reduction rules $\boxed{e, \mu \longrightarrow_{e} e, \mu, \Psi}$

$$\text{Switch}\ \frac{M, \mu \longrightarrow^{\text{ccd}}_{e} N, \mu, \Psi}{M, \mu \longrightarrow^{\text{cca}}_{e} N, \mu, \Psi} \qquad \text{Pure}\ \frac{M \longrightarrow^{\oslash}_{e} N}{M, \mu \longrightarrow^{\text{ccd}}_{e} N, \mu, \emptyset} \qquad \text{Ref}\ \frac{M, \mu \longrightarrow^{\bigcirc}_{e} N, \mu', \Psi}{M, \mu \longrightarrow^{\text{ccd}}_{e} N, \mu', \Psi}$$

$$\text{Cast/Succeed}\ \frac{\text{apply-cast}(v, c) = \text{just}\ (\Psi, v')}{v\langle c\rangle, \mu \longrightarrow^{\text{ccd}}_{e} v', \mu, \Psi} \qquad \text{Cast/fail}\ \frac{\text{apply-cast}(v, c) = \text{nothing}}{v\langle c\rangle, \mu \longrightarrow^{\text{ccd}}_{e} \texttt{error}, \mu, \emptyset}$$

$$\text{Cast/compose}\ \frac{}{M\langle c\rangle\langle d\rangle, \mu \longrightarrow^{\text{ccd}}_{e} M\langle c \fatsemi d\rangle, \mu, \emptyset}$$

$$\text{Cong}\ \frac{M, \mu \longrightarrow^{\text{cca}}_{e} N, \mu', \Psi}{F[M], \mu \longrightarrow^{\text{ccd}}_{e} F[N], \mu', \Psi} \qquad \text{CongErr}\ \frac{}{F[\texttt{error}], \mu \longrightarrow^{\text{ccd}}_{e} \texttt{error}, \mu, \emptyset}$$

$$\text{CongCast}\ \frac{M, \mu \longrightarrow^{\text{ccd}}_{e} N, \mu', \Psi}{M\langle c\rangle, \mu \longrightarrow^{\text{cca}}_{e} N\langle c\rangle, \mu', \Psi} \qquad \text{CongCastE}\ \frac{}{\texttt{error}\langle c\rangle, \mu \longrightarrow^{\text{cca}}_{e} \texttt{error}, \mu, \emptyset}$$

State reduction rules $\boxed{e, \mu, \Psi \longrightarrow e, \mu, \Psi}$

$$\text{ProgReduce}\ \frac{e, \mu \longrightarrow_{e} e', \mu, \Psi}{e, \mu, \emptyset \longrightarrow e', \mu, \Psi} \qquad \text{UpdateHeap}\ \frac{\begin{array}{c} T' = T \sqcap \mu(a)_{\text{rtti}} \quad T' \neq \mu(a)_{\text{rtti}} \\ \text{apply-cast}(\mu(a)_{\text{val}}, (\mu(a)_{\text{rtti}} \Rightarrow T')) = \text{just}\ (\Psi', v') \end{array}}{e, \mu, (a, T); \Psi \longrightarrow e, \mu(a \mapsto v' : T'), \Psi \oplus \Psi'}$$

$$\text{NoChange}\ \frac{T \sqcap \mu(a)_{\text{rtti}} = \mu(a)_{\text{rtti}}}{e, \mu, (a, T); \Psi \longrightarrow e, \mu, \Psi} \qquad \text{Error1}\ \frac{\begin{array}{c} T' = T \sqcap \mu(a)_{\text{rtti}} \quad T' \neq \mu(a)_{\text{rtti}} \\ \text{apply-cast}(\mu(a)_{\text{val}}, (\mu(a)_{\text{rtti}} \Rightarrow T')) = \text{nothing} \end{array}}{e, \mu, (a, T); \Psi \longrightarrow \texttt{error}, \mu, \Psi}$$

$$\text{Error2}\ \frac{T \nsim \mu(a)_{\text{rtti}}}{e, \mu, (a, T); \Psi \longrightarrow \texttt{error}, \mu, \Psi}$$

Fig. 4. Dynamic semantics for the space-efficient coercion calculus

$$a \mapsto \langle f \langle (((\iota\,;\,\mathsf{Int}!) \to (\mathsf{Int}?\,;\,\iota)))\rangle, a\rangle : (\mathsf{Int} \to \mathsf{Int}) \times \mathsf{Ref}\,((\mathsf{Int} \to \mathsf{Int}) \times \star)$$

Now the queue is empty, expression reduction will resume and the program will evaluate to 42.

6.1 Type Safety

A lot of the definitions and lemmas used to prove type safety are shared between λ_C^{ref} and λ_S^{ref}, so to highlight the important parts of the proof and to avoid duplication, they will be presented in the appendix only.

Definition 6.1 (Well-defined suspended casts queues). A queue of suspended casts Ψ is well-defined with respect to heap typing Σ, written $\Sigma \vdash \Psi$, iff $\forall a, T.\ \forall (a, T) \in \Psi$ implies $a \in \Sigma$.

The queue of suspended casts plays a role in typing as well as in reduction. More specifically, the typing of values in the heap depends on both the heap typing and the queue of suspended casts. We combine both using the following function, merge.

Definition 6.2 (Merging Heap Typing and Suspended Casts Queue).

$$\begin{array}{rcl} \text{merge}\ (\Sigma,\ \emptyset) & = & \Sigma \\ \text{merge}\ (\Sigma,\ ((a,T);\Psi)) & = & \text{merge}\ ((\Sigma(a \mapsto (T \sqcap \Sigma(a)))),\ \Psi) \quad \text{if } \Sigma(a) \sim T \\ \text{merge}\ (\Sigma,\ ((a,T);\Psi)) & = & \text{merge}\ (\Sigma,\ \Psi) \quad \text{if } \Sigma(a) \nsim T \end{array}$$

Definition 6.3 (Well-typed heaps). A heap μ is well-typed with respect to heap typing Σ and queue of suspended casts Ψ, written $\Sigma \mid \Psi \vdash \mu$, iff $\forall a, T.\ \Sigma(a) = T$ implies $\exists\, v\ s.t.$ merge $(\Sigma,\ \Psi) \mid \emptyset \vdash v : T$ and $\mu(a) = v : T$.

Lemma 6.4 (Populating the Queue). *If $\Sigma \vdash \Psi$, then, $\forall\ \Psi',\ s.t.\ \Sigma \vdash \Psi'$, merge $(\Sigma,\ (\Psi \oplus \Psi')) \sqsubseteq_p$ merge $(\Sigma,\ \Psi)$*

From Lemma C.4, we have the following corollary.

Corollary 6.5 (Heap cast). *If $\Sigma \vdash (a, T'); \Psi$ and $\Sigma \mid (a, T'); \Psi \vdash \mu$ and (merge $(\Sigma,\ \Psi)) \mid \emptyset \vdash v : T$ and $\mu(a) = v : T$, then address a can be cast to type* Ref $(T \sqcap T')$ *such that $\Sigma(a \mapsto (T \sqcap T')) \mid \Psi \oplus \Psi' \vdash \mu(a \mapsto v' : (T \sqcap T'))$ if such v' and Ψ' exist.*

Proof. By cases on the result of (apply-cast$(v, (T \Rightarrow (T \sqcap T'))))$:

nothing : We conclude with a cast error.

just (Ψ', v') : Let $\Sigma' = \Sigma(a \mapsto (T \sqcap T'))$. From $(T \sqcap T') \sqsubseteq T$ and $\Sigma \mid (a, T'); \Psi \vdash \mu$ we have $(T \sqcap T') \sqsubseteq \Sigma(a)$ which implies $\Sigma' \sqsubseteq_p \Sigma$. Also, by the definition of merge, we have merge $(\Sigma',\ \Psi) =$ merge $(\Sigma,\ ((a, T'); \Psi))$, so from that and Lemma 6.4, we have that merge $(\Sigma',\ (\Psi \oplus \Psi')) \sqsubseteq_p$ merge $(\Sigma',\ \Psi) \sqsubseteq_p$ merge $(\Sigma,\ ((a, T'); \Psi))$. Next, Lemma C.4 is applied to all values in the heap and to v', so we get (merge $(\Sigma',\ (\Psi \oplus \Psi'))) \mid \emptyset \vdash v' : (T \sqcap T')$, thus $\Sigma' \mid \Psi \oplus \Psi' \vdash \mu(a \mapsto v' : (T \sqcap T'))$.

□

Lemma 6.6 (Progress with suspended casts). *If $\Sigma \vdash \Psi$ and $\Sigma \mid \Psi \vdash \mu$ and $\Psi \neq \emptyset$ and merge $(\Sigma,\ \Psi) \mid \emptyset \vdash M : T$, then $\exists\, \mu'\ s.t.\ M, \mu, \Psi \longrightarrow M, \mu', \Psi'$*

Proof. Because Ψ is not empty, $\exists\, a, T, \Psi'.\ s.t.\ \Psi = (a, T); \Psi'$. By cases on how T and $\mu(a)_{\text{rtti}}$ are related, a step is taken as follows:

$T \sim \mu(a)_{\textbf{rtti}}$ **and** $T \sqcap \mu(a)_{\textbf{rtti}} \neq \mu(a)_{\textbf{rtti}}$**:** By cases on the result of (apply-cast$(\mu(a)_{\text{val}}, (\mu(a)_{\text{rtti}} \Rightarrow T)))$:

nothing: A step is taken via Error1.

just (v, Ψ'')**:** A step is taken via UpdateHeap.

$T \sim \mu(a)_{\mathbf{rtti}}$ **and** $T \sqcap \mu(a)_{\mathbf{rtti}} = \mu(a)_{\mathbf{rtti}}$**:** A step is taken via NoChange.
$T \nsim \mu(a)_{\mathbf{rtti}}$**:** A step is taken via Error2.

□

If the queue of suspended casts is empty, then the progress lemma is standard.

Lemma 6.7 (Progress with no suspended casts). *If* $\Sigma \mid \emptyset \vdash \mu$ *and* $\Sigma \mid \emptyset \vdash M : T$*, then either*

(1) M *is a value, or*
(2) $M = \texttt{error}$*, or*
(3) $\exists\, \Psi, N, \mu'\ s.t.\ M, \mu, \emptyset \longrightarrow N, \mu', \Psi$

From Lemmas 6.6 and 6.7, we assemble the full proof of progress by cases on whether the queue of suspended casts is empty:

Corollary 6.8 (Progress). *If* $\Sigma \vdash \Psi$ *and* $\Sigma \mid \Psi \vdash \mu$ *and merge* $(\Sigma,\ \Psi) \mid \emptyset \vdash M : T$*, then either*

(1) M *is a value, or*
(2) $M = \texttt{error}$*, or*
(3) $\exists\, N, \mu'\ s.t.\ M, \mu, \Psi \longrightarrow N, \mu', \Psi'$

Lemma 6.9 (Type preservation). *If* $\Sigma \vdash \Psi$ *and* $\Sigma \mid \Psi \vdash \mu$ *and merge* $(\Sigma,\ \Psi) \mid \emptyset \vdash M : T$ *and* $M, \mu, \Psi \longrightarrow N, \mu', \Psi'$*, then* $\exists\, \Sigma'\ s.t.\ \Sigma' \vdash \Psi'$ *and merge* $(\Sigma',\ \Psi') \mid \emptyset \vdash N : T$ *and* $\Sigma' \mid \Psi' \vdash \mu'$ *and* $\Sigma' \sqsubseteq_{p/e} \Sigma$ *and merge* $(\Sigma',\ \Psi') \sqsubseteq_{p/e}$ *merge* $(\Sigma,\ \Psi)$

From Corollary 6.8 and Lemma 6.9, we prove type safety.

Theorem 6.10 (Type safety). *If* $\Sigma \vdash \Psi$ *and* $\Sigma \mid \Psi \vdash \mu$ *and merge* $(\Sigma,\ \Psi) \mid \emptyset \vdash e : T$*, then either*

(1) e *diverges, or*
(2) $e = \texttt{error}$*, or*
(3) $\exists\, v, \Sigma', \mu'\ s.t.\ e, \mu, \Psi \longrightarrow^* v, \mu', \emptyset$ *and* $\Sigma' \mid \emptyset \vdash v : T$ *and* $\Sigma' \mid \emptyset \vdash \mu'$

The semantics and proof of type safety is fully mechanized in Agda proof assistant and it is available in the following URL:

https://github.com/deyaaeldeen/monotonic.

7 RELATED WORK

We conjecture that the space-efficient semantics for λ_S^{ref} that we define in this paper yields the same observable behavior as that of the non-space-efficient semantics λ_C^{ref}, which is the semantics of [Siek et al. 2015c] with some corrections. We define λ_C^{ref} in Appendix B.

In the following we discuss some additional related work regarding both monotonicity and space efficiency.

Monotonicity. Fähndrich and Leino [2003] present a statically checkable typestate system where objects may flow from less restrictive to more restrictive typestates, but not vice versa. Monotonic references is different in that they require run-time checks in partially typed code regions to maintain the monotonicity invariant.

Bierman et al. [2010] provide formal operational semantics for C♯ with the `dynamic` type added, named $\text{FC}_4^\sharp$. It uses an RTTI-based approach and subtype checks to implement casts. However, their approach is for a class-based object-oriented language as opposed to ML-like structural reference type that our work is based on. In particular, their work does not use the consistency relation and instead extends the implicit type conversion relation such that a type can be cast to `dynamic` if it can be cast to `object`.

Rastogi et al. [2015] and Swamy 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 where each object is instrumented with RTTI, but with respect to subtyping, treating $\star$ as a universal supertype, instead of with respect to the precision relation. Furthermore, they are different from monotonic references in that they inherit the overhead of dynamic typing because they compile to JavaScript. Moreover, in Safe TypeScript, the object's RTTI gets updated when a read field operation is performed. In monotonic references, reading from the heap does not change the RTTI.

Space efficiency. Siek and Wadler [2010] represent casts by threesomes and provide a composition operator of threesomes that uses the greatest lower bound function on types. Furthermore, they provide a compilation from threesomes to coercions and show how to compose them based on threesomes composition. Siek and Garcia [2012] define a space-efficient abstract machine for the gradually-typed lambda calculus based on coercions. Greenberg [2015] Provides a composition operator for manifest contracts, contracts that check stronger properties than simple types. However, reference types are not addressed in all aforementioned work.

8 CONCLUSION

Monotonic references is a design for gradually-typed mutable references that improves over the traditional design as they incur zero-overhead for reference operations in statically-typed code regions while maintaining the full expressiveness of gradual typing. However, prior work on monotonic references presented reduction semantics that writes cast expressions to the heap that reduce using a small-step reduction relation. It is not straightforward to implement such a process efficiently in a compiler and runtime system for gradually-typed languages. Furthermore, earlier work did not guarantee space-efficiency, a key property to ensure runtime efficiency in implementations of gradually-typed languages.

In this paper, we present novel dynamic semantics for monotonic references that is space-efficient and that only writes values to the heap. The former is accomplished by identifying a normal form for coercions with monotonic references and by extending the composition function accordingly. Moreover, in our dynamic semantics, we apply a cast to a value using a simple recursive function that suspends inner casts of references by putting them in a queue. This process is straightforward to implement in a runtime system for a gradually-typed language. Finally, we provide a type safety proof that is fully mechanized in Agda.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1763922.

REFERENCES

Christopher Anderson and Sophia Drossopoulou. 2003. BabyJ - From Object Based to Class Based Programming via Types. In *WOOD '03*, Vol. 82. Elsevier.

Gavin Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding dynamic types to C# *(ECOOP'10)*. Springer-Verlag, 25.

Manuel Fähndrich and K Rustan M Leino. 2003. Heap monotonic typestates. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*.

Robert Bruce Findler and Matthias Felleisen. 2002. *Contracts for Higher-Order Functions.* Technical Report NU-CCS-02-05. Northeastern University.

Michael Greenberg. 2015. Space-Efficient Manifest Contracts *(POPL '15)*. ACM, New York, NY, USA, 181–194. https://doi.org/10.1145/2676726.2676967

Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. 2006. Sage: Hybrid Checking for Flexible Specifications. In *Scheme and Functional Programming Workshop*. 93–104.

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

David Herman, Aaron Tomb, and Cormac Flanagan. 2007. Space-Efficient Gradual Typing. In *Trends in Functional Prog. (TFP)*. XXVIII.

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* 23, 2 (2010), 167–189.

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. Toward Efficient Gradual Typing for Structural Types via Coercions. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM.

Fabian Muehlboeck and Ross Tate. 2017. Sound Gradual Typing is Nominally Alive and Well. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 56 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133880

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript *(POPL '15)*. ACM, New York, NY, USA, 167–180. https://doi.org/10.1145/2676726.2676971

Jeremy G. Siek and Ronald Garcia. 2012. Interpretations of the Gradually-Typed Lambda Calculus. In *Scheme and Functional Programming Workshop*.

Jeremy G. Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the Design Space of Higher-Order Casts. In *European Symposium on Programming (ESOP)*. 17–31.

Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*. 81–92.

Jeremy G. Siek, Peter Thiemann, and Philip Wadler. 2015a. Blame and coercion: Together again for the first time. In *Conference on Programming Language Design and Implementation (PLDI)*.

Jeremy G. Siek and Michael M. Vitousek. 2013. Monotonic References for Gradual Typing. *CoRR* abs/1312.0694 (2013).

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015b. Refined Criteria for Gradual Typing. In *SNAPL: Summit on Advances in Programming Languages (LIPIcs: Leibniz International Proceedings in Informatics)*.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015c. Monotonic References for Efficient Gradual Typing. In *European Symposium on Programming (ESOP)*.

Jeremy G. Siek and Philip Wadler. 2010. Threesomes, with and without blame *(POPL)*. 365–376.

Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. 2014. Gradual Typing Embedded Securely in JavaScript. In *ACM Conference on Principles of Programming Languages (POPL)*.

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *Dynamic Languages Symposium*.

Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*. 1–16.

$$\text{Expressions} \quad e, M, N \quad ::= \quad k \mid \lambda x.\, e \mid e\, e \mid \langle e, e \rangle \mid \texttt{fst}\ e \mid \texttt{snd}\ e \mid \texttt{ref}\ e \mid !e \mid e := e$$

Fig. 5. GTLC$^+$ syntax

Types

$$\begin{array}{llll} \text{Base Types} & B & ::= & () \mid \mathsf{Bool} \mid \mathsf{Int} \\ \text{Injectables} & I, J & ::= & B \mid T \to T \mid T \times T \mid \mathsf{Ref}\ T \\ \text{Types} & T, S & ::= & \star \mid I \end{array}$$

Consistency $\boxed{T \sim T}$

$$\star \sim T \qquad T \sim \star \qquad B \sim B \qquad \frac{T_1 \sim T_2}{\mathsf{Ref}\ T_1 \sim \mathsf{Ref}\ T_2} \qquad \frac{T_1 \sim T_3 \quad T_2 \sim T_4}{T_1 \to T_2 \sim T_3 \to T_4} \qquad \frac{T_1 \sim T_3 \quad T_2 \sim T_4}{T_1 \times T_2 \sim T_3 \times T_4}$$

Shallow consistency $\boxed{T \smile T}$

$$\star \smile T \qquad T \smile \star \qquad B \smile B \qquad \frac{}{\mathsf{Ref}\ T_1 \smile \mathsf{Ref}\ T_2} \qquad \frac{}{T_1 \to T_2 \smile T_3 \to T_4} \qquad \frac{}{T_1 \times T_2 \smile T_3 \times T_4}$$

Type preciseness $\boxed{T \sqsubseteq T}$

$$T \sqsubseteq \star \qquad B \sqsubseteq B \qquad \frac{T_1 \sqsubseteq T_2}{\mathsf{Ref}\ T_1 \sqsubseteq \mathsf{Ref}\ T_2} \qquad \frac{T_1 \sqsubseteq T_3 \quad T_2 \sqsubseteq T_4}{T_1 \to T_2 \sqsubseteq T_3 \to T_4} \qquad \frac{T_1 \sqsubseteq T_3 \quad T_2 \sqsubseteq T_4}{T_1 \times T_2 \sqsubseteq T_3 \times T_4}$$

Static types $\boxed{\int T}$

$$\int B \qquad \frac{\int T}{\int \mathsf{Ref}\ T} \qquad \frac{\int T_1 \quad \int T_2}{\int T_1 \to T_2} \qquad \frac{\int T_1 \quad \int T_2}{\int T_1 \times T_2}$$

Meet operation (greatest lower bound) $\boxed{T \sqcap T = T}$

$$\star \sqcap T = T \sqcap \star = T \qquad B \sqcap B = B \qquad (T_1 \times T_2) \sqcap (T_3 \times T_4) = (T_1 \sqcap T_3) \times (T_2 \sqcap T_4)$$
$$\mathsf{Ref}\ T_1 \sqcap \mathsf{Ref}\ T_2 = \mathsf{Ref}\ (T_1 \sqcap T_2) \qquad (T_1 \to T_2) \sqcap (T_3 \to T_4) = (T_1 \sqcap T_3) \to (T_2 \sqcap T_4)$$

Fig. 6. Types and their operations

A TYPE SYSTEM OF GTLC$^+$

The syntax of the surface language, GTLC$^+$, is given in Figure 5. GTLC$^+$ has functions and pairs in addition to references. Figure 6 describes supported types and relations and functions on them. We use the symbol $\star$ to represent the dynamic type. Consistency and type preciseness relations are typical [Siek and Taha 2006; Siek et al. 2015b; Wadler and Findler 2009]. Shallow consistency [Siek and Wadler 2010] is needed to implement lazy error detection. The unary relation $\int$ holds for types that do not contain $\star$. Finally, the $\sqcap$ function computes the greatest lower bound of the input types.

As with any other gradually-typed language, GTLC$^+$ is compiled to a statically-typed cast calculus, by inserting casts. The cast calculus uses coercions [Henglein 1994] for cast specification. Figure 7 presents the syntax and the type system. Addresses a are required by rule WT-Addr to have a type Ref T where $\Sigma(a) \sqsubseteq T$. This restriction makes sure for each address a, $\forall r = a$ and $r : \mathsf{Ref}\ T$, that $\Sigma(a) \sim T$, i.e. the types of all references are always consistent with the type of the value in the heap. Expression forms for operations on references are unusual in that there are two forms for each operation, one to be used in fully static code regions and the other to be used in other regions [Siek et al. 2015c]. In particular, $!e$ and $e := e$ correspond to load and store machine

$$
\begin{array}{lllll}
\text{Expressions} & e, M, N & ::= & k \mid \lambda(x : T).\, e \mid e\, e \mid \langle e, e\rangle \mid \mathtt{fst}\ e \mid \mathtt{snd}\ e \mid e\langle c\rangle \mid \\
 & & & a \mid \mathtt{ref}\ e@T \mid !e \mid !e@T \mid e := e \mid e := e@T \mid \mathtt{error} \\
\text{Typing environment} & \Gamma & ::= & \emptyset \mid \Gamma, x : T \\
\text{Heap typing} & \Sigma & ::= & \emptyset \mid \Sigma, a : T
\end{array}
$$

Expression typing $\boxed{\Sigma \mid \Gamma \vdash M : T}$

$$\frac{}{\Sigma \mid \Gamma \vdash k : \textsc{type}(k)} \qquad \frac{(x : T) \in \Gamma}{\Sigma \mid \Gamma \vdash x : T} \qquad \frac{\Sigma \mid \Gamma \vdash M : T_1 \to T_2 \qquad \Sigma \mid \Gamma \vdash N : T_1}{\Sigma \mid \Gamma \vdash M\, N : T_2}$$

$$\frac{\Sigma \mid \Gamma, x : T_1 \vdash M : T_2}{\Sigma \mid \Gamma \vdash \lambda(x : T_1).\, M : T_1 \to T_2} \qquad \frac{\Sigma \mid \Gamma \vdash M : T_1 \qquad \Sigma \mid \Gamma \vdash N : T_2}{\Sigma \mid \Gamma \vdash \langle M, N\rangle : T_1 \times T_2}$$

$$\frac{\Sigma \mid \Gamma \vdash \langle M, N\rangle : T_1 \times T_2}{\Sigma \mid \Gamma \vdash \mathtt{fst}\ M : T_1} \qquad \frac{\Sigma \mid \Gamma \vdash \langle M, N\rangle : T_1 \times T_2}{\Sigma \mid \Gamma \vdash \mathtt{snd}\ M : T_2}$$

$$\frac{\Sigma \mid \Gamma \vdash M : T}{\Sigma \mid \Gamma \vdash \mathtt{ref}\ M@T : \mathsf{Ref}\ T} \qquad \frac{\Sigma \mid \Gamma \vdash M : \mathsf{Ref}\ T \qquad \wr T}{\Sigma \mid \Gamma \vdash !M : T}$$

$$\frac{\Sigma \mid \Gamma \vdash M : \mathsf{Ref}\ T \qquad \wr T}{\Sigma \mid \Gamma \vdash !M@T : T} \qquad \frac{\Sigma \mid \Gamma \vdash M : \mathsf{Ref}\ T \qquad \Sigma \mid \Gamma \vdash N : T \qquad \wr T}{\Sigma \mid \Gamma \vdash M := N : ()}$$

$$\frac{\Sigma \mid \Gamma \vdash M : \mathsf{Ref}\ T \qquad \Sigma \mid \Gamma \vdash N : T \qquad \wr T}{\Sigma \mid \Gamma \vdash M := N@T : ()}$$

$$\textsc{WT-Addr}\ \frac{\Sigma(a) \sqsubseteq T}{\Sigma \mid \Gamma \vdash a : \mathsf{Ref}\ T} \qquad \frac{}{\Sigma \mid \Gamma \vdash \mathtt{error} : T} \qquad \frac{\Sigma \mid \Gamma \vdash M : T \qquad c : T \Longrightarrow T'}{\Sigma \mid \Gamma \vdash M\langle c\rangle : T'}$$

Fig. 7. Syntax and type system of the coercion calculus

instructions and should be used in static code regions as they do not perform any checks. This is true because monotonic references have a unique canonical form which is just addresses, so unlike proxied references [Herman et al. 2007, 2010], there is no dynamic dispatch on whether the address is proxied. Also, by the typing rule for addresses WT-Addr, the type of a heap cell must be as precise as or more precise than all references to it, so it must be the case that the type of the value in the heap matches the type of the statically typed reference. On the other hand, $!e@T$ and $e := e@T$ are reserved for other code regions where the reference type is not static. The type annotation T records the type of the reference. The type of the value in the heap might be more precise than T, so a cast is needed between them. Asking the programmers to use both forms would be inconvenient. Fortunately, the compiler can automatically select which form to emit. Cast insertion, defined in Figure 8, is a type-directed compilation from GTLC$^+$ to the coercion calculus including compiling to the right form of a reference operation based on the type of the reference. Rules CI-SRead and CI-SWrite check if the reference type is static to compile the operation form to the corresponding efficient one. Rules CI-Read and CI-Write compile to the other forms. The CI-Allocation rule compiles the allocation form to one that records the reference type. The recorded type annotation is used to initialize the run-time type information (RTTI) of the heap cell. The other rules for typing and cast insertion are standard.

Cast insertion $\boxed{\Sigma \mid \Gamma \vdash M \hookrightarrow N : T}$

$$\frac{}{\Sigma \mid \Gamma \vdash k \hookrightarrow k : \textsc{type}(k)} \qquad \frac{(x : T) \in \Gamma}{\Sigma \mid \Gamma \vdash x \hookrightarrow x : T}$$

$$\frac{\Sigma \mid x : T_1, \Gamma \vdash M \hookrightarrow N : T_2}{\Sigma \mid \Gamma \vdash \lambda(x : T_1).\, M \hookrightarrow \lambda(x : T_1).\, N : T_1 \to T_2}$$

$$\frac{\Sigma \mid \Gamma \vdash M \hookrightarrow M' : T_1 \to T_2 \qquad \Sigma \mid \Gamma \vdash N \hookrightarrow N' : T_1'}{\Sigma \mid \Gamma \vdash M\, N \hookrightarrow M'\,(N'\langle T_1' \Rightarrow T_1\rangle) : T_2}$$

$$\frac{\Sigma \mid \Gamma \vdash M \hookrightarrow M' : \star \qquad \Sigma \mid \Gamma \vdash N \hookrightarrow N' : T}{\Sigma \mid \Gamma \vdash M\, N \hookrightarrow (M'\langle \star \Rightarrow (\star \Rightarrow \star)\rangle)\,(N'\langle T \Rightarrow \star\rangle) : \star}$$

$$\frac{\Sigma \mid \Gamma \vdash M \hookrightarrow M' : T_1 \qquad \Sigma \mid \Gamma \vdash N \hookrightarrow N' : T_2}{\Sigma \mid \Gamma \vdash \langle M, N\rangle \hookrightarrow \langle M', N'\rangle : T_1 \times T_2}$$

$$\frac{\Sigma \mid \Gamma \vdash M \hookrightarrow N : T_1 \times T_2}{\Sigma \mid \Gamma \vdash \mathtt{fst}\ M \hookrightarrow \mathtt{fst}\ N : T_1} \qquad \frac{\Sigma \mid \Gamma \vdash M \hookrightarrow N : T_1 \times T_2}{\Sigma \mid \Gamma \vdash \mathtt{snd}\ M \hookrightarrow \mathtt{snd}\ N : T_2}$$

$$\frac{\Sigma \mid \Gamma \vdash M \hookrightarrow N : \star}{\Sigma \mid \Gamma \vdash \mathtt{fst}\ M \hookrightarrow \mathtt{fst}\ (N\langle \star \Rightarrow \star \times \star\rangle) : \star} \qquad \frac{\Sigma \mid \Gamma \vdash M \hookrightarrow N : \star}{\Sigma \mid \Gamma \vdash \mathtt{snd}\ M \hookrightarrow \mathtt{snd}\ (N\langle \star \Rightarrow \star \times \star\rangle) : \star}$$

$$\textsc{CI-Allocation}\ \frac{\Sigma \mid \Gamma \vdash M \hookrightarrow N : T}{\Sigma \mid \Gamma \vdash \mathtt{ref}\ M \hookrightarrow \mathtt{ref}\ N@T : \mathsf{Ref}\ T} \qquad \textsc{CI-SRead}\ \frac{\Sigma \mid \Gamma \vdash M \hookrightarrow N : \mathsf{Ref}\ T \qquad \wr T}{\Sigma \mid \Gamma \vdash !M \hookrightarrow !N : T}$$

$$\textsc{CI-Read}\ \frac{\Sigma \mid \Gamma \vdash M \hookrightarrow N : \mathsf{Ref}\ T}{\Sigma \mid \Gamma \vdash !M \hookrightarrow !N@T : T} \qquad \frac{\Sigma \mid \Gamma \vdash M \hookrightarrow N : \star}{\Sigma \mid \Gamma \vdash !M \hookrightarrow !(N\langle \star \Rightarrow \mathsf{Ref}\ \star\rangle)@\star : \star}$$

$$\textsc{CI-SWrite}\ \frac{\Sigma \mid \Gamma \vdash M \hookrightarrow M' : \mathsf{Ref}\ T \qquad \Sigma \mid \Gamma \vdash N \hookrightarrow N' : T \qquad \wr T}{\Sigma \mid \Gamma \vdash M := N \hookrightarrow M' := N' : ()}$$

$$\textsc{CI-Write}\ \frac{\Sigma \mid \Gamma \vdash M \hookrightarrow M' : \mathsf{Ref}\ T \qquad \Sigma \mid \Gamma \vdash N \hookrightarrow N' : T' \qquad T \sim T' \qquad T \neq T'}{\Sigma \mid \Gamma \vdash M := N \hookrightarrow M' := (N'\langle T' \Rightarrow T\rangle)@T' : ()}$$

$$\frac{\Sigma \mid \Gamma \vdash M \hookrightarrow M' : \star \qquad \Sigma \mid \Gamma \vdash N \hookrightarrow N' : T}{\Sigma \mid \Gamma \vdash M := N \hookrightarrow (M'\langle \star \Rightarrow \mathsf{Ref}\ \star\rangle) := (N'\langle T' \Rightarrow \star\rangle)@\star : ()}$$

Fig. 8. Compiling GTLC$^+$ to the coercion calculus

B DYNAMIC SEMANTICS OF $\lambda_C^{\mathbf{ref}}$

Monotonic references were first introduced in terms of an abstract machine mechanized in Isabelle/HOL [Siek and Vitousek 2013]. Later, Siek et al. [2015c] presented a reduction semantics for monotonic references based on coercions along with the addition of blame tracking. However, The latter had a few minor problems that λ_C^{ref} fix.

- The grammar for casted values[1] does not allow casts on other casted values. However, the rule CastRef1 in the cast reduction relation reads a casted value from the heap and then

[1] λ_C^{ref} uses the term *delayed casts* instead to avoid confusion with values that have casts on them

$$
\begin{array}{lllll}
\text{Coercions} & c, d & ::= & \iota \mid I! \mid I? \mid c \to d \mid c \times d \mid \mathsf{Ref}\ T \mid \bot^{I,J} \\
\text{Inert coercions} & c\uparrow & ::= & I! \mid c \to d \\
\text{Active coercions} & c\downarrow & ::= & \iota \mid I? \mid c \times d \mid \mathsf{Ref}\ T \mid \bot^{I,J}
\end{array}
$$

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

$$
\frac{}{\iota : T \Longrightarrow T} \qquad \frac{}{I! : I \Longrightarrow \star} \qquad \frac{}{I? : \star \Longrightarrow I} \qquad \frac{I \nsim J}{\bot^{I,J} : I \Longrightarrow J}
$$

$$
\frac{c : T_1' \Longrightarrow T_1 \quad d : T_2 \Longrightarrow T_2'}{c \to d : T_1 \to T_2 \Longrightarrow T_1' \to T_2'} \qquad \frac{c : T_1 \Longrightarrow T_1' \quad d : T_2 \Longrightarrow T_2'}{c \times d : T_1 \times T_2 \Longrightarrow T_1' \times T_2'} \qquad \frac{}{\mathsf{Ref}\ T' : \mathsf{Ref}\ T \Longrightarrow \mathsf{Ref}\ T'}
$$

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

$$
\begin{array}{lll}
(T \Rightarrow T) = \iota & & (T_1 \to T_2 \Rightarrow T_1' \to T_2') = (T_1' \Rightarrow T_1) \to (T_2 \Rightarrow T_2') \\
(I \Rightarrow \star) = I! \qquad (\star \Rightarrow I) = I? & & (T_1 \times T_2 \Rightarrow T_1' \times T_2') = (T_1 \Rightarrow T_1') \times (T_2 \Rightarrow T_2') \\
(I \Rightarrow J) = \bot^{I,J} \qquad \text{if } I \not\sim J & & (\mathsf{Ref}\ T \Rightarrow \mathsf{Ref}\ T') = \mathsf{Ref}\ T'
\end{array}
$$

Fig. 9. Simple coercions and their operations

wraps it in another cast expression that is written to the heap. λ_C^{ref} fixes this issue by changing the grammar of delayed casts to allow casts on delayed casts (Figure 10).

- A casted value could reduce to an expression that has an inner error using a congruence rule. Such expression is not a casted value and can not be written to the heap. The state reduction relation has a rule to handle reduction to an error expression but does not have one to handle reduction to expressions that have an inner error. λ_C^{ref} fixes this issue by updating the Error rule to make sure the result delayed cast could be decomposed into a context and an error expression.
- The type preservation lemma, the second item in Lemma 2, relates the old heap typing and the new heap typing with the precision relation on heap typing (Definition 1). However, this does not account for the fact that the heap could grow by creating new references. λ_C^{ref} fixes this issue by recognizing that the heap typing could move along two orthogonal dimensions, it could become either more precise (Definition C.1) or larger (Definition C.2). The type preservation lemma is updated accordingly (Lemma C.18).

The variant of the semantics of λ_C^{ref}[2] and associated type safety proof is fully mechanized in Agda proof assistant. Figure 9 presents simple coercions and their operations. Inert coercions, denoted $c\uparrow$, form values without checking at the time of the cast. Injection and function coercions are inert coercions. On the other hand, all other coercions do not form values and are called active coercions, denoted $c\downarrow$.

Figure 10 presents the definition of values, delayed casts, heaps, evolving heaps, and frames. Figures 11 and 12 presents the reduction rules for the cast calculus with simple coercions. The symbol $\oslash$ indicates a reduction relation that does not relate heaps. The symbol $\bigcirc$ indicates a reference reduction relation.

C TYPE SAFETY FOR λ_C^{ref}

We prove type safety for the corrected version of λ_C^{ref} (defined in Appendix B).

[2]The difference is state reduction relation has more verbose rules in the mechanized version

Run-Time Structures

Values	v	::=	$k \mid \lambda(x : T).\, e \mid \langle v, v\rangle \mid a \mid v\langle c\uparrow\rangle$
Delayed Casts	dc	::=	$v \mid dc\langle c\rangle \mid \langle dc, dc\rangle$
Heap	μ	::=	$\emptyset \mid \mu(a \mapsto v : T)$
Evolving Heap	ν	::=	$\mu \mid \nu(a \mapsto dc : T)$
Frames	F	::=	$\square\, e \mid v\,\square \mid \langle \square, e\rangle \mid \langle v, \square\rangle \mid \texttt{fst}\ \square \mid \texttt{snd}\ \square \mid \square\langle c\rangle \mid$ $\texttt{ref}\ \square@T \mid\ !\square@T \mid\ !\square \mid \square := e@T \mid v := \square@T$
Evaluation Contexts	$\mathcal{E}$	::=	$\emptyset \mid \mathrm{F}, \mathcal{E}$

Fig. 10. Run-Time structures for the simple coercion calculus

Pure coercion reduction rules $\boxed{e \longrightarrow_c^{\not\square} e}$

$$v\langle \iota\rangle \longrightarrow_c^{\not\square} v \qquad v\langle \bot^{I,J}\rangle \longrightarrow_c^{\not\square} \texttt{error}$$
$$v\langle I!\rangle\langle J?\rangle \longrightarrow_c^{\not\square} v\langle I \Rightarrow J\rangle \qquad \langle v_1, v_2\rangle\langle c \times d\rangle \longrightarrow_c^{\not\square} \langle v_1\langle c\rangle, v_2\langle d\rangle\rangle$$

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

$$\textsc{PureCast}\ \frac{M \longrightarrow_c^{\not\square} N}{M, \nu \longrightarrow_c N, \nu}$$

$$\textsc{CastRef1}\ \frac{\nu(a) = dc : T_1 \qquad T_1 \sim T_2 \qquad T_1 \sqcap T_2 \neq T_1}{a\langle \texttt{Ref}\ T_2\rangle, \nu \longrightarrow_c a, \nu(a \mapsto dc\langle T_1 \Rightarrow (T_1 \sqcap T_2)\rangle : T_1 \sqcap T_2)}$$

$$\textsc{CastRef2}\ \frac{\nu(a)_{\text{rtti}} \sim T \qquad \nu(a)_{\text{rtti}} \sqcap T = \nu(a)_{\text{rtti}}}{a\langle \texttt{Ref}\ T\rangle, \nu \longrightarrow_c a, \nu} \qquad \textsc{CastRef3}\ \frac{\nu(a)_{\text{rtti}} \nsim T}{a\langle \texttt{Ref}\ T\rangle, \nu \longrightarrow_c \texttt{error}, \nu}$$

$$\textsc{Cong}_c\ \frac{M, \nu \longrightarrow_c N, \nu'}{\mathrm{F}[M], \nu \longrightarrow_c \mathrm{F}[N], \nu'} \qquad \textsc{CongErr}_c\ \frac{}{\mathcal{E}[\texttt{error}], \nu \longrightarrow_c \texttt{error}, \nu}$$

Fig. 11. Dynamic semantics for casts in the simple coercion calculus

We first recognize that the heap typing can move along two orthogonal dimensions, it could become either more precise (Definition C.1) or its domain can grow (Definition C.2)[3]. Combining both relations results in one, $\sqsubseteq_{p/e}$, that fully captures how the heap typing changes after each step, called Heap Typing Progress (Definition C.3).

Definition C.1 (Precision relation on heap typings). $\Sigma' \sqsubseteq_p \Sigma$ iff $\mathrm{dom}(\Sigma') = \mathrm{dom}(\Sigma)$ and $\Sigma(a) = T$ implies $\Sigma'(a) = T'$ where $T' \sqsubseteq T$.

Definition C.2 (Extension relation on heap typings). $\Sigma \sqsubseteq_e \Sigma'$ iff $\mathrm{dom}(\Sigma) \subseteq \mathrm{dom}(\Sigma')$ and $\Sigma(a) = \Sigma'(a)$.

Definition C.3 (Progress relation for heap typings). $\Sigma' \sqsubseteq_{p/e} \Sigma$ if $\Sigma' \sqsubseteq_p \Sigma$ or $\Sigma \sqsubseteq_e \Sigma'$.

Lemma C.4 asserts that expression typing is preserved when moving to a more precise heap typing. Similarly, Lemma C.5 asserts that expression typing is also preserved when moving to a larger heap typing. Corollary C.6 combines both lemmas and asserts that expression typing is preserved when the heap typing changes with respect to the $\sqsubseteq_{p/e}$ relation.

[3]We ignore garbage collection and assume heaps can not become smaller

Pure reduction rules $\boxed{e \longrightarrow_e^{\oslash} e}$

$$\begin{array}{llll} \texttt{fst}\ \langle v_1, v_2\rangle & \longrightarrow_e^{\oslash} v_1 & (\lambda(x:T).\,M)\,v & \longrightarrow_e^{\oslash} [x := v]M \\ \texttt{snd}\ \langle v_1, v_2\rangle & \longrightarrow_e^{\oslash} v_2 & (v_1\langle c \to d\rangle)\,v_2 & \longrightarrow_e^{\oslash} (v_1\,(v_2\langle c\rangle))\langle d\rangle \end{array}$$

Reference operations reduction rules $\boxed{e, \mu \longrightarrow_e^{\ocirc} e, \nu}$

$$\texttt{ref}\ v@T, \mu \longrightarrow_e^{\ocirc} a, \mu(a \mapsto v : T) \quad \text{if } a \notin \text{dom}(\mu) \qquad \text{(NSEAlloc)}$$

$$!a, \mu \longrightarrow_e^{\ocirc} \mu(a)_{\text{val}}, \mu \qquad \text{(Read)}$$

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

$$a := v, \mu \longrightarrow_e^{\ocirc} \text{Unit}, \mu(a \mapsto v : \mu(a)_{\text{rtti}}) \qquad \text{(Update)}$$

$$a := v@T, \mu \longrightarrow_e^{\ocirc} \text{Unit}, \mu(a \mapsto v\langle T \Rightarrow \mu(a)_{\text{rtti}}\rangle : \mu(a)_{\text{rtti}}) \qquad \text{(DynUpdate)}$$

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

$$\textsc{Pure}\ \frac{M \longrightarrow_e^{\oslash} N}{M, \mu \longrightarrow N, \mu} \qquad \textsc{Mono}\ \frac{M, \mu \longrightarrow_e^{\ocirc} N, \nu}{M, \mu \longrightarrow N, \nu}$$

$$\textsc{Cong}\ \frac{M, \nu \longrightarrow N, \nu'}{\mathrm{F}[M], \mu \longrightarrow \mathrm{F}[N], \nu} \qquad \textsc{CongErr}\ \frac{}{\mathcal{E}[\texttt{error}], \mu \longrightarrow \texttt{error}, \nu}$$

$$\textsc{Cast}\ \frac{e, \mu \longrightarrow_c e', \nu}{e, \mu \longrightarrow e', \nu} \qquad \textsc{Error}\ \frac{\nu(a) = dc : T \qquad dc, \nu \longrightarrow_c \mathcal{E}[\texttt{error}], \nu'}{e, \nu \longrightarrow \texttt{error}, \nu'}$$

$$\textsc{NoRTTIChange}\ \frac{\nu(a) = dc : T \qquad dc, \nu \longrightarrow_c dc', \nu' \qquad \nu'(a)_{\text{rtti}} = T}{e, \nu \longrightarrow e, \nu'(a \mapsto dc' : T)}$$

$$\textsc{RTTIChanged}\ \frac{\nu(a) = dc : T \qquad dc, \nu \longrightarrow_c dc', \nu' \qquad \nu'(a)_{\text{rtti}} \neq T}{e, \nu \longrightarrow e, \nu'}$$

Fig. 12. Dynamic semantics for the simple coercion calculus

Lemma C.4 (Strengthening wrt. the heap typing precision). *If* $\Sigma \mid \Gamma \vdash e : T$ *and* $\Sigma' \sqsubseteq_p \Sigma$, *then* $\Sigma' \mid \Gamma \vdash e : T$.

Proof. For the case of addresses: From $\Sigma' \sqsubseteq_p \Sigma$ and the WT-addr rule and transitivity of $\sqsubseteq$, we have $\Sigma'(a) \sqsubseteq T$. Therefore $\Sigma' \mid \Gamma \vdash a : T$. □

Lemma C.5 (Weakening wrt. the heap typing extension). *If* $\Sigma \mid \Gamma \vdash e : T$ *and* $\Sigma \sqsubseteq_e \Sigma'$, *then* $\Sigma' \mid \Gamma \vdash e : T$.

Proof. For the case of addresses: we have $\Sigma'(a) \sqsubseteq T$ from $\Sigma \sqsubseteq_e \Sigma$ and the WT-addr rule. Therefore $\Sigma' \mid \Gamma \vdash a : T$. □

Corollary C.6 (Weakening wrt. the heap typing progress). *If* $\Sigma \mid \Gamma \vdash e : T$ *and* $\Sigma' \sqsubseteq_{p/e} \Sigma$, *then* $\Sigma' \mid \Gamma \vdash e : T$.

Proof. By applying Lemmas C.4 and C.5 in the cases of $\sqsubseteq_p$ and $\sqsubseteq_e$ respectively. □

Heaps are well-typed when the delayed casts inside respect heap typing.

Definition C.7 (Well-typed heaps). A heap v is well-typed with respect to heap typing Σ, written $\Sigma \vdash v$, iff $\forall a, T.\ \Sigma(a) = T$ implies $\exists\ dc\ s.t.\ \Sigma \mid \emptyset \vdash dc : T$ and $v(a) = dc : T$.

From Lemma C.4, we have the following corollary.

Corollary C.8 (Heap cast). *If $\Sigma \vdash v$ and $v(a) = dc : T$ and $T' \sqsubseteq T$, then a can be cast to* Ref T' *such that $\Sigma(a \mapsto T') \vdash v(a \mapsto (dc\langle T \Rightarrow T'\rangle) : T')$.*

Proof. Let $\Sigma' = \Sigma(a \mapsto T')$ and $dc' = dc\langle T \Rightarrow T'\rangle$. From $T' \sqsubseteq T$ and $\Sigma \vdash v$ we have $T' \sqsubseteq \Sigma(a)$ which implies $\Sigma' \sqsubseteq_p \Sigma$. Next, Lemma C.4 is applied to all delayed casts in the heap and to dc', so we get $\Sigma' \mid \emptyset \vdash dc' : T'$, thus $\Sigma' \vdash v(a \mapsto dc' : T')$. □

Similarly, from Lemma C.5, we have the following corollary.

Corollary C.9 (Heap extension). *If $\Sigma \vdash \mu$ and $\Sigma \mid \emptyset \vdash v : T$ and $a \notin dom(\Sigma)$, then v can be added at a such that $\Sigma(a \mapsto T) \vdash \mu(a \mapsto v : T)$.*

Proof. Let $\Sigma' = \Sigma(a \mapsto T)$. We have $\Sigma \sqsubseteq_e \Sigma'$ from $a \notin \mathrm{dom}(\Sigma)$. Next, Lemma C.5 is applied to all values in the heap and to v to get $\Sigma' \mid \emptyset \vdash v : T$. We add v to the heap at address a, thus $\Sigma' \vdash \mu(a \mapsto v : T)$. □

Lemma C.10 shows that coercions are either inert or active.

Lemma C.10 (Inert and active coercions). *$\forall c : T \Longrightarrow T'$, $c \uparrow$ or $c \downarrow$*

Active coercions do not create values, so it better be the case that they can take a step. Lemma C.11 asserts this fact:

Lemma C.11 (Active cast progress). *If $\Sigma \vdash v$ and $\Sigma \mid \emptyset \vdash v : T$ and $c : T \Longrightarrow T'$ and $c \downarrow$, then $\exists\ e, v'\ s.t.\ v\langle c\rangle, v \longrightarrow_c e, v'$*

Proof. By case analysis on $c \downarrow$. In the case of monotonic reference coercion, v must be some address a, and $v\langle \texttt{Ref}\ T'\rangle$ takes a step via $\longrightarrow_c$. In particular:

CastRef1 if $T' \sim v(a)_{\text{rtti}}$ and $T' \sqcap v(a)_{\text{rtti}} \neq v(a)_{\text{rtti}}$.
CastRef2 if $T' \sim v(a)_{\text{rtti}}$ and $T' \sqcap v(a)_{\text{rtti}} = v(a)_{\text{rtti}}$.
CastRef3 if $T' \nsim v(a)_{\text{rtti}}$.

□

The following lemma proves that a delayed cast can take step if it is not a value.

Lemma C.12 (Reducible delayed cast progress). *If $\Sigma \vdash v$ and $\Sigma \mid \emptyset \vdash dc : T$ and dc is not a value, then $\exists\ e, v'\ s.t.\ dc, v \longrightarrow_c e, v'$*

Proof. By case analysis on dc.

Case v Contradiction by the premise that dc is not a value.
Case $dc'\langle c\rangle$ By cases on whether dc' is a value
  Case *yes* We know that dc is not a value, so it must be the case that c is neither an injection or a function coercion, so Lemma C.11 is applied.
  Case *no* The induction hypothesis is applied to dc and a step is taken via Cong_c.
Case $\langle dc_1, dc_2\rangle$ It must be the case that either or both projections are not values, so the induction hypothesis is applied to first projection that is not a value and a step is taken via Cong_c.

□

However this is not true for arbitrary cast expressions because inert coercions form values.

Corollary C.13 (Cast progress). *If* $\Sigma \vdash v$ *and* $\Sigma \mid \emptyset \vdash v : T$ *and* $c : T \Longrightarrow T'$*, then either*

(1) $v\langle c \rangle$ *is a value, or*
(2) $\exists\, e, v'\ s.t.\ v\langle c \rangle, v \longrightarrow_c e, v'$

Proof. By Lemma C.10, c is either $c\uparrow$ or $c\downarrow$. In the case of the former, we conclude that $v\langle c \rangle$ is a value. Otherwise, Lemma C.11 is applied. □

If a delayed cast took a reduction step, the result expression should be another delayed cast or an erroneous expression.

Lemma C.14 (Reducible delayed cast preservation). *If* $\Sigma \vdash v$ *and* $\Sigma \mid \emptyset \vdash dc : T$ *and* $\Sigma' \mid \emptyset \vdash e : T$ *and* $\Sigma' \vdash v'$ *and* $dc, v \longrightarrow_c e, v'$*, then either*

(1) e is a delayed cast, or
(2) $e = \mathcal{E}[\texttt{error}]$

we prove progress when the heap contains at least one delayed cast that is not a value.

Lemma C.15 (Progress with evolving heap). *If* $\Sigma \vdash v$ *and* v *is not a normal heap and* $\Sigma \mid \emptyset \vdash M : T$*, then* $\exists\, v'\ s.t.\ M, v \longrightarrow M, v'$

Proof. Because v is not normal, then $\exists\, a, dc\ s.t.\ v(a)_{\text{val}} = dc$ and dc is not a *value*. By Lemma C.12, dc takes a step to an expression e. By Lemma C.14, there are two cases:

Case $e = \mathcal{E}[\texttt{error}]$: A step is taken via Error.
Case e **is a delayed cast** : By cases on whether $v(a)_{\text{rtti}} = v'(a)_{\text{rtti}}$
  Case $v(a)_{\textbf{rtti}} = v'(a)_{\textbf{rtti}}$ A step is taken via NoRTTIChange
  Case $v(a)_{\textbf{rtti}} \neq v'(a)_{\textbf{rtti}}$ A step is taken via RTTIChanged

□

Progress in the case of a normal heap is standard.

Lemma C.16 (Progress with normal heap). *If* $\Sigma \vdash \mu$ *and* $\Sigma \mid \emptyset \vdash M : T$*, then either*

(1) M is a value, or
(2) $M = \texttt{error}$*, or*
(3) $\exists\, N, v\ s.t.\ M, \mu \longrightarrow N, v$

Proof. By Lemma C.13 and standard induction on the typing derivation. □

From Lemmas C.15 and C.16, we assemble the full proof of progress by cases on whether the heap is in a normal state:

Corollary C.17 (Progress). *If* $\Sigma \vdash v$ *and* $\Sigma \mid \emptyset \vdash M : T$*, then either*

(1) M is a value, or
(2) $M = \texttt{error}$*, or*
(3) $\exists\, N, v'\ s.t.\ M, v \longrightarrow N, v'$

Lemma C.18 (Type preservation). *If* $\Sigma \vdash v$ *and* $\Sigma \mid \emptyset \vdash M : T$ *and* $M, v \longrightarrow N, v'$*, then* $\exists\, \Sigma'\ s.t.\ \Sigma' \mid \emptyset \vdash N : T$ *and* $\Sigma' \vdash v'$ *and* $\Sigma' \sqsubseteq_{p/e} \Sigma$

From Corollary C.16 and Lemma C.18, we prove type safety.

Theorem C.19 (Type safety). *If* $\Sigma \vdash v$ *and* $\Sigma \mid \emptyset \vdash e : T$*, then either*

(1) e diverges, or
(2) $e = \texttt{error}$*, or*
(3) $\exists\, v, \Sigma', v'\ s.t.\ e, v \longrightarrow^* v, v'$ *and* $\Sigma' \mid \emptyset \vdash v : T$ *and* $\Sigma' \vdash v'$