Space-Efficient Gradual Typing

David Herman[1], Aaron Tomb[2], and Cormac Flanagan[2]

[1] Northeastern University
[2] University of California, Santa Cruz

Abstract

Gradual type systems offer a smooth continuum between static and dynamic typing by permitting the free mixture of typed and untyped code. The runtime systems for these languages—and other languages with hybrid type checking—typically enforce function types by dynamically generating function proxies. This approach can result in unbounded growth in the number of proxies, however, which drastically impacts space efficiency and destroys tail recursion.

We present an implementation strategy for gradual typing that is based on *coercions* instead of function proxies, and which combines adjacent coercions to limit their space consumption. We prove bounds on the space consumed by coercions as well as soundness of the type system, demonstrating that programmers can safely mix typing disciplines without incurring unreasonable overheads. Our approach also detects certain errors earlier than prior work.

1 GRADUAL TYPING FOR SOFTWARE EVOLUTION

Dynamically typed languages have always excelled at exploratory programming. Languages such as Lisp, Scheme, Smalltalk, and JavaScript support quick early prototyping and incremental development without the overhead of documenting (often-changing) structural invariants as types. For large applications, however, static type systems have proven invaluable. They are crucial for understanding and enforcing key program invariants and abstractions, and they catch many errors early in the development process.

Given these different strengths, it is not uncommon to encounter the following scenario: a programmer builds a prototype in a dynamically-typed scripting language, perhaps even in parallel to a separate, official software development process with an entire team. The team effort gets mired in process issues and over-engineering, and the programmer's prototype ends up getting used in production. Before long, this hastily conceived prototype grows into a full-fledged production system, but without the structure or guarantees provided by static types. The system becomes unwieldy; QA can't produce test cases fast enough and bugs start cropping up that no one can track down. Ultimately, the team decides to port the application to a statically typed language, requiring a complete rewrite of the entire system.

The cost of cross-language migration is huge and often insupportable. But the scenario above is avoidable. Several languages combine static and dynamic typing, among them Boo [7], Visual Basic.NET [19], Sage [16], and PLT Scheme [24]. This approach of *hybrid typing*, where types are enforced with a combination of

static and dynamic checks, has begun to gain attention in the research community [5, 16, 22, 24, 19, 2]. Recently, Siek and Taha [22, 23] coined the slogan *gradual typing* for this important application of hybrid typing: the ability to implement both partially-conceived prototypes and mature, production systems in the same programming language by gradually introducing type discipline. Gradual typing offers the possibility of continuous software evolution from prototype to product, thus avoiding the huge costs of language migration.

Our recent experience in the working group on the JavaScript [8] language specification provides a more concrete example. JavaScript is a dynamically-typed functional programming language that is widely used for scripting user interactions in web browsers, and is a key technology in Ajax applications [15]. The enormous popularity of the language is due in no small part to its low barrier to entry; anyone can write a JavaScript program by copying and pasting code from one web page to another. Its dynamic type system and fail-soft runtime semantics allow programmers to produce something that *seems* to work with a minimum of effort. The increasing complexity of modern web applications, however, has motivated the addition of a static type system. The working group's intention is not to abandon the dynamically-typed portion of the language—because of its usefulness and to retain backwards compatibility—but rather to allow typing disciplines to interact via a hybrid system that supports gradual typing.[1]

1.1 The Cost of Gradual Typing

Gradually-typed languages support both statically-typed and dynamically-typed code, and include runtime checks (or type casts) at the boundaries between these two typing disciplines, to guarantee that dynamically-typed code cannot violate the invariants of statically-typed code. To illustrate this idea, consider the following code fragment, which passes an untyped variable x into a variable y of type `Int`:

$$\texttt{let}\ x = \texttt{true}\ \texttt{in}\ \ldots\ \texttt{let}\ y : \texttt{Int} = x\ \texttt{in}\ \ldots$$

During compilation, the type checker inserts a dynamic type cast $\langle \texttt{Int} \rangle$ to enforce the type invariant on y; at run-time, this cast detects the attempted type violation:

$$\begin{array}{ll} & \texttt{let}\ x = \texttt{true}\ \texttt{in}\ \ldots\ \texttt{let}\ y : \texttt{Int} = (\langle \texttt{Int} \rangle\ x)\ \texttt{in}\ \ldots \\ \longrightarrow^* & \mathsf{Error} : \textit{“failed cast”} \end{array}$$

Unfortunately, even these simple, first-order type checks can result in unexpected costs, as in the following example, where a programmer has added some type annotations to a previously untyped program:

$$\begin{array}{l} \textit{even} \qquad\qquad\quad = \lambda n{:}\texttt{Int}.\ \mathsf{if}\ (n = 0)\ \mathsf{then}\ \texttt{true}\ \ \mathsf{else}\ \textit{odd}\ (n-1) \\ \textit{odd} : \texttt{Int} \to \texttt{Bool} = \lambda n{:}\texttt{Int}.\ \mathsf{if}\ (n = 0)\ \mathsf{then}\ \texttt{false}\ \mathsf{else}\ \textit{even}\ (n-1) \end{array}$$

[1] The JavaScript specification is a work in progress, but gradual typing is a key design goal [9].

This program seems innocuous, but suffers from a space leak. Since *even* is dynamically typed, the result of each call to *even* $(n-1)$ must be cast to `Bool`, resulting in unbounded growth in the control stack and destroying tail recursion.

Additional complications arise when first-class functions cross the boundaries between typing disciplines. In general, it is not possible to check if an untyped function satisfies a particular static type. A natural solution is to wrap the function in a proxy that, whenever it is applied, casts its argument and result values appropriately, ensuring that the function is only observed with its expected type. This proxy-based approach is used heavily in recent literature [11, 12, 16, 18, 22], but has serious consequences for space efficiency.

As a simple example, consider the following program in continuation-passing style, where both mutually recursive functions take a continuation argument k, but only one of these arguments is annotated with a precise type:

$$\begin{array}{ll} \mathit{even} = \lambda n{:}\texttt{Int}.\,\lambda k{:}(? \to ?). & \mathsf{if}\ (n=0)\ \mathsf{then}\ (k\ \texttt{true}) \quad \mathsf{else}\ \mathit{odd}\ (n-1)\ k \\ \mathit{odd}\ = \lambda n{:}\texttt{Int}.\,\lambda k{:}(\texttt{Bool} \to \texttt{Bool}). & \mathsf{if}\ (n=0)\ \mathsf{then}\ (k\ \texttt{false})\ \mathsf{else}\ \mathit{even}\ (n-1)\ k \end{array}$$

Here, the recursive calls to *odd* and *even* quietly cast the continuation argument k with higher-order casts $\langle \texttt{Bool} \to \texttt{Bool} \rangle$ and $\langle ? \to ? \rangle$, respectively. This means that the function argument k is wrapped in an additional function proxy at each recursive call!

The flexibility promised by gradual typing can only be achieved if programmers are free to decide how precisely to type various parts of a program. This flexibility is lost if adding type annotations can accidentally trigger asymptotic changes in its space consumption. In short, existing implementation techniques for gradual typing suffer from unacceptable space leaks.

1.2 Space-Efficient Gradual Typing

We present an implementation strategy for gradual typing that overcomes these problems. Our approach hinges on the simple insight that when proxies accumulate at runtime, they often contain redundant information. In the higher-order example above, the growing chain of function proxies contains only two distinct components, $\texttt{Bool} \to \texttt{Bool}$ and $? \to ?$, which could be merged to the simpler but equivalent $\texttt{Bool} \to \texttt{Bool}$ proxy.

Type casts behave like *error projections* [10], which are closed under composition. However, the syntax of casts does not always compose; for example, there is no cast c such that $\langle c \rangle\ e = \langle \texttt{Int} \rangle\ \langle \texttt{Bool} \rangle\ e$. Furthermore, projections are idempotent, which should allow us to eliminate duplicate casts. For example, $\langle \texttt{Bool} \to \texttt{Bool} \rangle\ \langle \texttt{Bool} \to \texttt{Bool} \rangle\ e = \langle \texttt{Bool} \to \texttt{Bool} \rangle\ e$. But such transformations are inconvenient, if not impossible, with a representation of higher-order type casts as functions.

Our formalization instead leverages Henglein's *coercion calculus* [17], which provides a syntax for projections, called coercions, which are closed under a composition operator. This allows us to combine adjacent coercions in order to eliminate redundant information and thus guarantee clear bounds on space consumption. By

eagerly combining coercions, we can also detect certain errors immediately as soon as a function cast is applied; in contrast, prior approaches would not detect these errors until the casted function is invoked.

Our approach is applicable to many hybrid-typed languages [12, 13, 16] that use function proxies and hence are prone to space consumption problems. For clarity, we formalize our approach for the simply-typed λ-calculus with references. Of course, gradual typing is not restricted to such simple type systems: the Sage language [16] incorporates gradual typing as part of a very expressive type system with polymorphic functions, type operators, first-class types, dependent types, general refinement types, etc. Concurrently, Siek and Taha [22] developed gradual typing for the simpler language $\lambda^{?}_{\to}$, which we use as the basis for this presentation.

The presentation of our results proceeds as follows. The following section reviews the syntax and type system of $\lambda^{?}_{\to}$. Section 3 introduces the coercion algebra underlying our approach. Section 4 describes how we compile source programs into a target language with explicit coercions. Section 5 provides an operational semantics for that target language, and Section 6 proves bounds on the space consumption. Section 7 extends our approach to detect errors earlier and provide better coverage. The last two sections place our work into context.

2 GRADUALLY-TYPED LAMBDA CALCULUS

This section reviews the gradually-typed λ-calculus $\lambda^{?}_{\to}$. This language is essentially the simply-typed λ-calculus extended with the type ? to represent dynamic types; it also includes mutable reference cells to demonstrate the gradual typing of assignments.

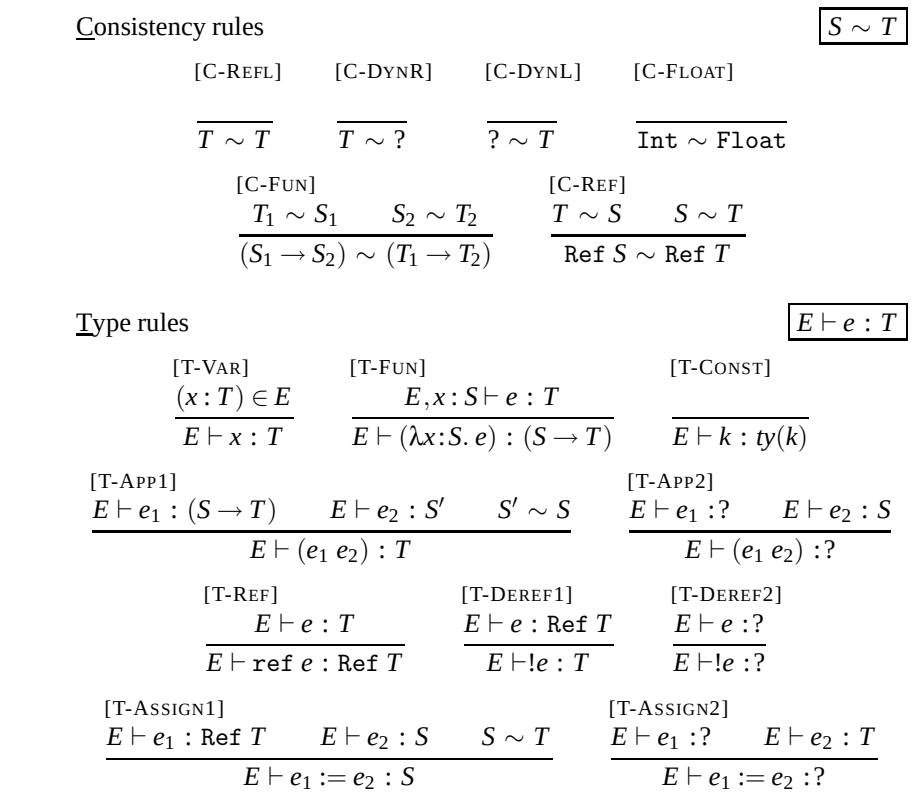
$$\begin{array}{lll} \textit{Terms:} & e & ::= k \mid x \mid \lambda x{:}T.\, e \mid e\ e \mid \texttt{ref}\ e \mid\ !e \mid e := e \\ \textit{Types:} & S,T & ::= B \mid T \to T \mid ? \mid \texttt{Ref}\ T \end{array}$$

Terms include the usual constants, variables, abstractions, and applications, as well as reference allocation, dereference, and assignment. Types include the dynamic type ?, function types $T \to T$, reference types $\texttt{Ref}\ T$, and some collection of ground or base types B (such as `Int` or `Float`).

The $\lambda^{?}_{\to}$ type system is a little unusual in that it is based on an intransitive *consistency* relation $S \sim T$ instead of the more conventional, transitive subtyping relation $S <: T$. Any type is consistent with the type ?, from which it follows that, for example, $\texttt{Bool} \sim ?$ and $? \sim \texttt{Int}$. However, booleans cannot be used directly as integers, which is why the consistency relation is not transitively closed. We do not assume the consistency relation is symmetric, since a language might, for example, allow coercions from integers to floats but not vice-versa.

The consistency relation is defined in Figure 1. Rules [C-DynL] and [C-DynR] allow all coercions to and from type ?. The rule [C-Float] serves as an example of asymmetry by allowing coercion from `Int` to `Float` but not the reverse. The rule [C-Fun] is reminiscent of the contravariant/covariant rule for function subtyping.

Figure 1: Source Language Type System



We extend the invariant reference cells of $\lambda^{?}_{\to}$ to allow coercion from `Ref` S to `Ref` T via rule [C-REF], provided S and T are symmetrically consistent. Unlike functions, reference cells do not distinguish their output ("read") type from their input ("write") type, so coercion must be possible in either direction. For example, the two reference types `Ref Int` and `Ref ?` are consistent.

Figure 1 also presents the type rules for the source language, which are mostly standard. Notice the presence of two separate rules for procedure application. Rule [T-APP1] handles the case where the operator is statically-typed as a function; in this case, the argument may have any type consistent with the function's domain. Rule [T-APP2] handles the case where the operator is dynamically-typed, in which case the argument may be of any type. The two rules for assignment follow an analogous pattern, accepting a consistent type when the left-hand side is known to have type `Ref` T, and any type when the left-hand side is dynamically-typed. Similarly, dereference expressions only produce a known type when the argument has a reference type.

3 COERCIONS

To achieve a space-efficient implementation, we compile source programs into a target language with explicit type casts, which allow expressions of one type to be used at any consistent type. Our representation of casts is based on *coercions*, drawn from Henglein's theory of dynamic typing [17]. The key benefit of coercions over prior proxy-based representations is that they are *combinable*; if two coercions are wrapped around a function value, then they can be safely combined into a single coercion, thus reducing the space consumption of the program without changing its semantic behavior.

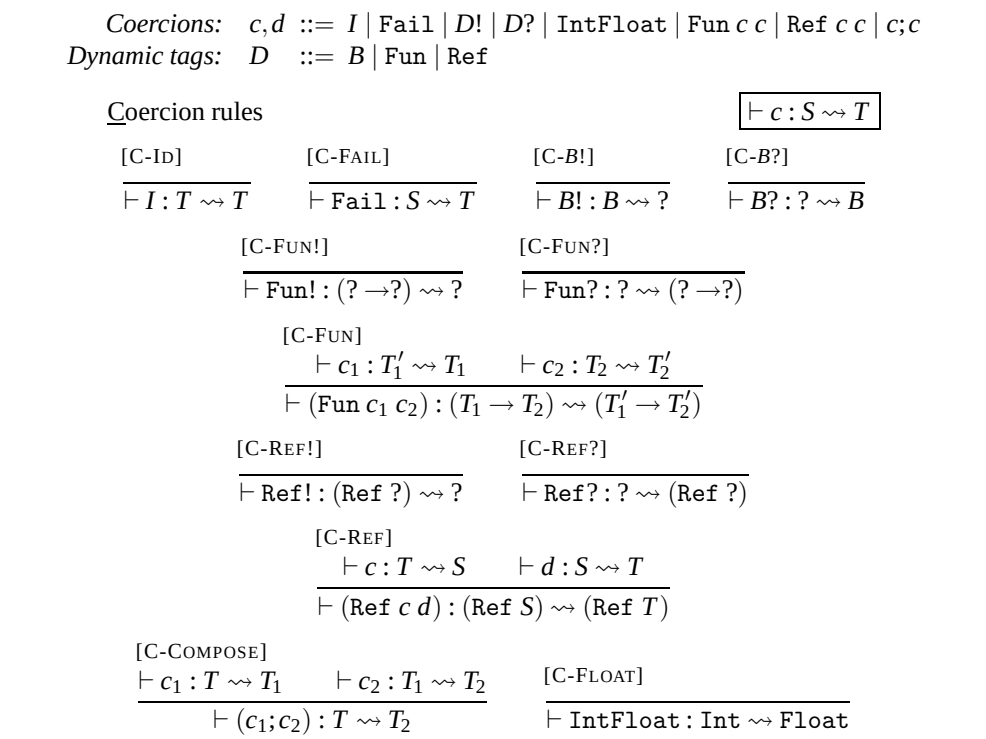
The coercion language and its typing rules are both defined in Figure 2. The coercion judgment $\vdash c : S \rightsquigarrow T$ states that coercion c serves to coerce values from type S to type T. The identity coercion I (of type $\vdash c : T \rightsquigarrow T$ for any T) always succeeds. Conversely, the failure coercion `Fail` always fails. For each dynamic type tag D there is an associated tagging coercion $D!$ that produces values of type ?, and a corresponding check-and-untag coercion $D?$ that takes values of type ?. Thus, for example, we have $\vdash \texttt{Int!} : \texttt{Int} \rightsquigarrow ?$ and $\vdash \texttt{Int?} : ? \rightsquigarrow \texttt{Int}$.

The function checking coercion `Fun?` converts a value of type ? to have the dynamic function type $? \rightarrow ?$. If a more precise function type is required, this value can be further coerced via a function coercion `Fun` c d, where c coerces function arguments and d coerces results. For example, the coercion (`Fun Int? Int!`) coerces from $? \rightarrow ?$ to $\texttt{Int} \rightarrow \texttt{Int}$, by untagging function arguments (via `Int?`) and tagging function results (via `Int!`). Reference coercions also contain two components: the first for coercing values put into the reference cell; the second for coercing values read from the cell. Finally, the coercion $c;d$ represents coercion composition, *i.e.*, the coercion c followed by coercion d.

This coercion language is sufficient to translate between all consistent types: if types S and T are consistent, then the following partial function $\mathit{coerce}(S,T)$ is defined and returns the appropriate coercion between these types.

$$
\begin{array}{rcl}
\mathit{coerce} : \mathit{Type} \times \mathit{Type} & \rightarrow & \mathit{Coercion} \\
\\
\mathit{coerce}(T,T) & = & I \\
\mathit{coerce}(B,?) & = & B! \\
\mathit{coerce}(?,B) & = & B? \\
\mathit{coerce}(\texttt{Int},\texttt{Float}) & = & \texttt{IntFloat} \\
\\
\mathit{coerce}(S_1 \rightarrow S_2, T_1 \rightarrow T_2) & = & \texttt{Fun}\ \mathit{coerce}(T_1,S_1)\ \mathit{coerce}(S_2,T_2) \\
\mathit{coerce}(?, T_1 \rightarrow T_2) & = & \texttt{Fun?};\mathit{coerce}(? \rightarrow ?, T_1 \rightarrow T_2) \\
\mathit{coerce}(T_1 \rightarrow T_2, ?) & = & \mathit{coerce}(T_1 \rightarrow T_2, ? \rightarrow ?);\texttt{Fun!} \\
\\
\mathit{coerce}(\texttt{Ref}\ S, \texttt{Ref}\ T) & = & \texttt{Ref}\ \mathit{coerce}(T,S)\ \mathit{coerce}(S,T) \\
\mathit{coerce}(?, \texttt{Ref}\ T) & = & \texttt{Ref?};\mathit{coerce}(\texttt{Ref}\ ?, \texttt{Ref}\ T) \\
\mathit{coerce}(\texttt{Ref}\ T, ?) & = & \mathit{coerce}(\texttt{Ref}\ T, \texttt{Ref}\ ?);\texttt{Ref!}
\end{array}
$$

Figure 2: Coercion Language and Type Rules



Coercing a type T to itself produces the identity coercion I. Coercing base types B to type ? requires a tagging coercion $B!$, and coercing ? to a base type B requires a runtime check $B?$. Function coercions work by coercing their domain and range types. The type ? is coerced to a function type via a two-step coercion: first the value is checked to be a function and then coerced from the dynamic function type $? \rightarrow ?$ to $T_1 \rightarrow T_2$. Dually, typed functions are coerced to type ? via coercion to a dynamic function type followed by the function tag `Fun!`. Coercing a `Ref` S to a `Ref` T entails coercing all writes from T to S and all reads from S to T. Coercing reference types to and from ? is analogous to function coercion.

Lemma 1 (Well-typed coercions).

1. *$S \sim T$ iff $coerce(S,T)$ is defined.*
2. *If $c = coerce(S,T)$ then $\vdash c : S \rightsquigarrow T$.*

Proof. Inductions on the derivations of $S \sim T$ and $coerce(S,T)$. □

Figure 3: Target Language Syntax and Type Rules

$$
\begin{array}{rrcl}
\textit{Terms:} & s,t & ::= & x \mid u \mid t\ t \mid \texttt{ref}\ t \mid\ !t \mid t := t \mid \langle c \rangle\ t \\
\textit{Values:} & v & ::= & u \mid \langle c \rangle\ u \\
\textit{Uncoerced values:} & u & ::= & \lambda x{:}T.\ t \mid k \mid a \\
\textit{Stores:} & \sigma & ::= & \emptyset \mid \sigma[a := v] \\
\textit{Typing environments:} & E & ::= & \emptyset \mid E, x : T \\
\textit{Store typings:} & \Sigma & ::= & \emptyset \mid \Sigma, a : T
\end{array}
$$

Type rules $\boxed{E;\Sigma \vdash t : T \qquad E;\Sigma \vdash \sigma}$

[T-VAR]
$$\frac{(x:t) \in E}{E;\Sigma \vdash x : T}$$

[T-FUN]
$$\frac{E, x:S;\Sigma \vdash t : T}{E;\Sigma \vdash (\lambda x{:}S.\ t) : (S \to T)}$$

[T-APP]
$$\frac{E;\Sigma \vdash t_1 : (S \to T) \qquad E;\Sigma \vdash t_2 : S}{E;\Sigma \vdash (t_1\ t_2) : T}$$

[T-REF]
$$\frac{E;\Sigma \vdash t : T}{E;\Sigma \vdash \texttt{ref}\ t : \texttt{Ref}\ T}$$

[T-DEREF]
$$\frac{E;\Sigma \vdash t : \texttt{Ref}\ T}{E;\Sigma \vdash !t : T}$$

[T-ASSIGN]
$$\frac{E;\Sigma \vdash t_1 : \texttt{Ref}\ T \qquad E;\Sigma \vdash t_2 : T}{E;\Sigma \vdash t_1 := t_2 : T}$$

[T-CONST]
$$\frac{}{E;\Sigma \vdash k : ty(k)}$$

[T-CAST]
$$\frac{\vdash c : S \rightsquigarrow T \qquad E;\Sigma \vdash t : S}{E;\Sigma \vdash \langle c \rangle\ t : T}$$

[T-ADDR]
$$\frac{(a:T) \in \Sigma}{E;\Sigma \vdash a : \texttt{Ref}\ T}$$

[T-STORE]
$$\frac{dom(\sigma) = dom(\Sigma) \qquad \forall a \in dom(\sigma).\ E;\Sigma \vdash \sigma(a) : \Sigma(a)}{E;\Sigma \vdash \sigma}$$

4 TARGET LANGUAGE AND CAST INSERTION

During compilation, we both type check the source program and insert explicit type casts where necessary. The target language of this cast insertion process is essentially the same as the source, except that it uses explicit casts of the form $\langle c \rangle\ t$ as the only mechanism for connecting terms of type ? and terms of other types. For example, the term $\langle \texttt{Int?} \rangle\ x$ has type `Int`, provided that x has type ?. The language syntax and type rules are defined in Figure 3, and are mostly straightforward. The language also includes addresses a which refer to a global store σ, and the store typing environment Σ maps addresses to types.

The process of type checking and inserting coercions is formalized via the *cast insertion judgment*:

$$E \vdash e \hookrightarrow t : T$$

Here, the type environment E provides types for free variables, e is the original source program, t is a modified version of the original program with additional coercions, and T is the inferred type for t. The rules defining the cast insertion judgment are shown in Figure 4, and they rely on the partial function *coerce* to compute

Figure 4: Cast Insertion Rules

Cast insertion rules $\boxed{E \vdash e \hookrightarrow t : T}$

$$\text{[C-VAR]}\ \frac{(x:T) \in E}{E \vdash x \hookrightarrow x : T} \qquad \text{[C-CONST]}\ \frac{}{E \vdash k \hookrightarrow k : ty(k)} \qquad \text{[C-FUN]}\ \frac{E, x:S \vdash e \hookrightarrow t : T}{E \vdash (\lambda x{:}S.\, e) \hookrightarrow (\lambda x{:}S.\, t) : (S \to T)}$$

$$\text{[C-APP1]}\ \frac{E \vdash e_1 \hookrightarrow t_1 : (S \to T) \qquad E \vdash e_2 \hookrightarrow t_2 : S' \qquad c = coerce(S', S)}{E \vdash e_1\ e_2 \hookrightarrow (t_1\ (\langle c \rangle\ t_2)) : T}$$

$$\text{[C-APP2]}\ \frac{E \vdash e_1 \hookrightarrow t_1 : ? \qquad E \vdash e_2 \hookrightarrow t_2 : S' \qquad c = coerce(S', ?)}{E \vdash e_1\ e_2 \hookrightarrow ((\langle \texttt{Fun?} \rangle\ t_1)\ (\langle c \rangle\ t_2)) : ?}$$

$$\text{[C-REF]}\ \frac{E \vdash e \hookrightarrow t : T}{E \vdash \texttt{ref}\ e \hookrightarrow \texttt{ref}\ t : \texttt{Ref}\ T} \qquad \text{[C-DEREF1]}\ \frac{E \vdash e \hookrightarrow t : \texttt{Ref}\ T}{E \vdash !e \hookrightarrow !t : T} \qquad \text{[C-DEREF2]}\ \frac{E \vdash e \hookrightarrow t : ?}{E \vdash !e \hookrightarrow !(\langle \texttt{Ref?} \rangle\ t) : ?}$$

$$\text{[C-ASSIGN1]}\ \frac{E \vdash e_1 \hookrightarrow t_1 : \texttt{Ref}\ S \qquad E \vdash e_2 \hookrightarrow t_2 : T \qquad c = coerce(T, S)}{E \vdash e_1 := e_2 \hookrightarrow (t_1 := (\langle c \rangle\ t_2)) : S}$$

$$\text{[C-ASSIGN2]}\ \frac{E \vdash e_1 \hookrightarrow t_1 : ? \qquad E \vdash e_2 \hookrightarrow t_2 : T \qquad c = coerce(T, ?)}{E \vdash e_1 := e_2 \hookrightarrow ((\langle \texttt{Ref?} \rangle\ t_1) := (\langle c \rangle\ t_2)) : ?}$$

coercions between types. For example, rule [C-APP1] compiles an application expression where the operator has a function type $S \to T$ by casting the argument expression from to type S. Rule [C-APP2] handles the case where the operator is dynamically-typed by inserting a $\langle \texttt{Fun?} \rangle$ check to ensure the operator evaluates to a function and casting the argument to type ? to yield tagged values. Rules [C-REF] and [C-DEREF1] handle typed reference allocation and dereference. Rule [C-DEREF2] handles dynamically-typed dereference by inserting a runtime $\langle \texttt{Ref?} \rangle$ check. Rule [C-ASSIGN1] handless statically-typed assignment, casting the right-hand side to the expected type of the reference cell, and [C-ASSIGN2] handles dynamically-typed assignment, casting the left-hand side with a runtime $\langle \texttt{Ref?} \rangle$ check and the right-hand side with a tagging coercion to type ?.

Compilation succeeds on all well-typed source programs, and produces only well-typed target programs.

Theorem 2 (Well-typed cast insertion). *For all E, e, and T, the following statements are equivalent:*

1. $E \vdash e : T$
2. $\exists t$ *such that* $E \vdash e \hookrightarrow t : T$ *and* $E; \emptyset \vdash t : T$

Proof. Inductions on the cast insertion and source language typing derivations. □

5 OPERATIONAL SEMANTICS

We now consider how to implement the target language in a manner that limits the space consumed by coercions. The key idea is to combine adjacent casts to eliminate redundant information while preserving the semantic behavior of programs.

Figure 5 provides the definitions and reduction rules for the target language, using a small-step operational semantics with evaluation contexts. For simplicity, we present our results here in a substitution semantics.[2] The grammar of evaluation contexts C is defined to prevent nesting of adjacent casts. This restriction allows the most important reduction rule, [E-CCAST], to ensure that adjacent casts in the program term are always merged.

In order to maintain bounds on their size, coercions are maintained normalized throughout evaluation according to the following rules:

$$
\begin{array}{rcl@{\qquad}rcl}
I;c & = & c & D!;D? & = & I \\
c;I & = & c & D!;D'? & = & \texttt{Fail} \quad \text{if } D \neq D' \\
\texttt{Fail};c & = & \texttt{Fail} & (\texttt{Fun}\ c_1\ c_2);(\texttt{Fun}\ d_1\ d_2) & = & \texttt{Fun}\ (d_1;c_1)\ (c_2;d_2) \\
c;\texttt{Fail} & = & \texttt{Fail} & (\texttt{Ref}\ c_1\ c_2);(\texttt{Ref}\ d_1\ d_2) & = & \texttt{Ref}\ (d_1;c_1)\ (c_2;d_2)
\end{array}
$$

This normalization is applied in a transitive, compatible manner whenever the rule [E-CCAST] is applied, thus bounding the size of coercions during evaluation.

Most of the remaining reduction rules are straightforward. Rules [E-BETA], [E-NEW], [E-DEREF], and [E-ASSIGN] are standard. The rule [E-PRIM] relies on a type-indexed family of functions $\delta_T : \mathit{Term} \times \mathit{Term} \to \mathit{Term}$ to define the semantics of constant functions. We assume each $\delta_{S \to T}$ to be well-typed and, for simplicity, defined for all constants $k : S \to T$ and arguments $v : S$. Rule [E-CAPP] applies function casts by casting the function argument and result. Rule [E-CDEREF] casts the result of reading a cell and [E-CASSIGN] casts the value written to a cell and casts the value again to the expected output type. Rules [E-ID] and [E-FCAST] respectively perform the identity and float coercions, and are restricted to non-cast contexts to prevent overlap with [E-CCAST].

Evaluation satisfies the usual preservation and progress lemmas.

Theorem 3 (Soundness of evaluation). *If* $\emptyset;\emptyset \vdash t : T$ *then either*

1. $t, \emptyset$ *diverges,*
2. $t, \emptyset \longrightarrow^* C[\langle \texttt{Fail} \rangle\ u], \sigma$ *or*
3. $t, \emptyset \longrightarrow^* v, \sigma$ *and* $\exists \Sigma$ *such that* $\emptyset;\Sigma \vdash v : T$ *and* $\emptyset;\Sigma \vdash \sigma$.

Proof. Via standard subject reduction and progress lemmas in the style of Wright and Felleisen [24]. □

[2] For a discussion of more detailed models of space usage, see Section 9.

Figure 5: Operational Semantics

$$
\begin{array}{rrl}
\textit{Evaluation Contexts:} & C & ::= \bullet \mid (C\ t) \mid (v\ C) \mid \texttt{ref}\ C \mid\ !C \mid C := t \mid v := C \mid \langle c \rangle\ D \\
\textit{Nested Contexts:} & D & ::= \bullet \mid (C\ t) \mid (v\ C) \mid \texttt{ref}\ C \mid\ !C \mid C := t \mid v := C
\end{array}
$$

$$
\begin{array}{rcll}
C[(\lambda x{:}S.\ t)\ v], \sigma & \longrightarrow & C[t[x := v]], \sigma & \text{[E-BETA]} \\
C[\texttt{ref}\ v], \sigma & \longrightarrow & C[a], \sigma[a := v] \quad \text{for } a \notin \mathit{dom}(\sigma) & \text{[E-NEW]} \\
C[!a], \sigma & \longrightarrow & C[\sigma(a)], \sigma & \text{[E-DEREF]} \\
C[a := v], \sigma & \longrightarrow & C[v], \sigma[a := v] & \text{[E-ASSIGN]} \\
C[k\ v], \sigma & \longrightarrow & C[\delta_{ty(k)}(k, v)], \sigma & \text{[E-PRIM]} \\
C[(\langle \texttt{Fun}\ c\ d \rangle\ u)\ v], \sigma & \longrightarrow & C[\langle d \rangle\ (u\ (\langle c \rangle\ v))], \sigma & \text{[E-CAPP]} \\
C[!(\langle \texttt{Ref}\ c\ d \rangle\ a)], \sigma & \longrightarrow & C[\langle d \rangle\ !a], \sigma & \text{[E-CDEREF]} \\
C[(\langle \texttt{Ref}\ c\ d \rangle\ a) := v], \sigma & \longrightarrow & C[\langle d \rangle\ (a := \langle c \rangle\ v)], \sigma & \text{[E-CASSIGN]} \\
C[\langle I \rangle\ u], \sigma & \longrightarrow & C[u], \sigma \quad \text{if } C \neq C'[\langle c \rangle\ \bullet] & \text{[E-ID]} \\
C[\langle \texttt{IntFloat} \rangle\ n], \sigma & \longrightarrow & C[\mathit{nearestFloat}(n)], \sigma \quad \text{if } C \neq C'[\langle c \rangle\ \bullet] & \text{[E-FCAST]} \\
C[\langle c \rangle\ (\langle d \rangle\ t)], \sigma & \longrightarrow & C[\langle d; c \rangle\ t], \sigma & \text{[E-CCAST]}
\end{array}
$$

6 SPACE EFFICIENCY

We now consider how much space coercions consume at runtime, beginning with an analysis of how much space each individual coercion can consume.

6.1 Space Consumption

The size of a coercion $\mathit{size}(c)$ is defined as the size of its abstract syntax tree representation. When two coercions are sequentially composed and normalized during evaluation, the size of the normalized, composed coercion may of course be larger than either of the original coercions. In order to reason about the space required by such composed coercions, we introduce a notion of the *height* of a coercion:

$$
\begin{array}{l}
\mathit{height}(I) = \mathit{height}(\texttt{Fail}) = \mathit{height}(D!) = \mathit{height}(D?) = 1 \\
\mathit{height}(\texttt{Ref}\ c\ d) = \mathit{height}(\texttt{Fun}\ c\ d) = 1 + \mathit{max}(\mathit{height}(c), \mathit{height}(d)) \\
\mathit{height}(c; d) = \mathit{max}(\mathit{height}(c), \mathit{height}(d))
\end{array}
$$

Notably, the height of a composed coercion is bounded by the maximum height of its constituents. In addition, normalization never increases the height of a coercion. Thus, the height of any coercion created during program evaluation is never larger than the height of some coercion in the original elaborated program.

Furthermore, this bound on the height of each coercion in turn guarantees a bound on the coercion's size, according to the following lemma. In particular, even though the length of a coercion sequence $c_1; \ldots; c_n$ does not contribute to its height, the restricted structure of well-typed, normalized coercions constrains the length (and hence size) of such sequences.

Lemma 4. *For all well-typed normalized coercions c,* $\mathit{size}(c) \leq 5(2^{\mathit{height}(c)} + 1)$.

Proof. Induction on c. Assume $c = c_1;\ldots;c_n$, where each c_i is not a sequential composition.

Suppose some $c_i = \texttt{Ref}\ d_1\ d_2$. So $\vdash c_i : \texttt{Ref}\ S \rightsquigarrow \texttt{Ref}\ T$. Hence c_i can be preceded only by `Ref?`, which must be the first coercion in the sequence, and similarly can be followed only by `Ref!`, which must be the last coercion in the sequence. Thus in the worst case $c = \texttt{Ref?};\texttt{Ref}\ d_1\ d_2;\texttt{Ref!}$ and $size(c) = 5 + size(d_1) + size(d_2)$. Applying the induction hypothesis to the sizes of d_1 and d_2 yields:

$$size(c) \leq 5 + 2(5(2^{height(c)-1} - 1)) = 5(2^{height(c)} - 1)$$

The case for $\texttt{Fun}\ d_1\ d_2$ is similar. The coercions I and `Fail` can only appear alone. Finally, coercions of the form $D?;D!$ are valid. However, composition of a coercion c matching this pattern with one of the other valid coercions is either ill-typed or triggers a normalization that yields a coercion identical to c. □

In addition, the height of any coercion created during cast insertion is bounded by the height of some type in the typing derivation (where the height of a type is the height of its abstract syntax tree representation).

Lemma 5. *If $c = coerce(S,T)$, then $height(c) \leq max(height(S), height(T))$.*

Proof. Induction on the structure of $coerce(S,T)$. □

Theorem 6 (Size of coercions). *For any e, c such that*

1. $\emptyset \vdash e \hookrightarrow t : T$ *and*
2. $t, \emptyset \longrightarrow^* t', \sigma$ *and*
3. *c occurs in* (t', σ),

$\exists S$ in the derivation of $\emptyset \vdash e \hookrightarrow t : T$ such that $size(c) \leq 5(2^{height(S)} - 1)$.

Proof. Induction on the length of the reduction sequence, using Lemma 4; the base case is by induction on the compilation derivation, using Lemma 5. □

We now bound the total cost of maintaining coercions in the space-efficient semantics. We define the size of a program state inductively as the sum of the sizes of its components. In order to construct a realistic measure of the store, we count only those cells that an idealized garbage collector would consider live by restricting the *size* function to the domain of reachable addresses.

$$\begin{aligned}
&size(t, \sigma) = size(t) + size(\sigma|_{reachable(t)}) \\
&size(\sigma) = \textstyle\sum_{a \in dom(\sigma)}(1 + size(\sigma(a))) \\
&size(k) = size(a) = size(x) = 1 \\
&size(\lambda x{:}T.\ t) = 1 + size(T) + size(t) \\
&size(\texttt{ref}\ t) = size(!t) = 1 + size(t) \\
&size(t_1 := t_2) = size(t_1\ t_2) = 1 + size(t_1) + size(t_2) \\
&size(\langle c \rangle\ t) = 1 + size(c) + size(t)
\end{aligned}$$

To show that coercions occupy bounded space in the model, we compare the size of program states in reduction sequences to program states in an "oracle" semantics where coercions require no space. The oracular measure $size_{\mathrm{OR}}$ is defined similarly to *size*, but without a cost for maintaining coercions; that is, $size_{\mathrm{OR}}(c) = 0$. The following theorem then bounds the fraction of the program state occupied by coercions in the space-efficient semantics.

Theorem 7 (Space consumption). *If* $\emptyset \vdash e \hookrightarrow t : T$ *and* $t, \emptyset \longrightarrow^* t', \sigma$, *then there exists some* S *in the derivation of* $\emptyset \vdash e \hookrightarrow t : T$ *such that* $size(t', \sigma) \in O(2^{height(S)} \cdot size_{\mathrm{OR}}(t', \sigma))$.

Proof. During evaluation, the [E-CCAST] rule prevents nesting of adjacent coercions in any term in the evaluation context, redex, or store. Thus the number of coercions in the program state is proportional to the size of the program state. By Theorem 6 the size of each coercion is in $O(2^{height(S)})$ for the largest S in the typing of e. □

6.2 Tail recursion

Theorem 7 has the important consequence that coercions do not affect the control space consumption of tail-recursive programs. For example, the *even* and *odd* functions mentioned in the introduction now consume constant space. This important property is achieved by always combining adjacent coercions on the stack via the [E-CCAST] rule. This section sketches three implementation techniques for this rule.

Coercion-passing style This approach adds an extra argument to every procedure, representing the result coercion. Tail calls coalesce but do not perform this coercion, instead passing it along to the next function.

Trampoline A trampoline [14] is a well-known technique for implementing tail-recursive languages where tail calls are implemented by returning a thunk to a top-level loop. Tail-recursive functions with coercions return both a thunk and a coercion to the driver loop, which accumulates and coalesces returned coercions.

Continuation marks Continuation marks [6] allow programs to annotate continuation frames with arbitrary data. When a marked frame performs a tail call, the subsequent frame can inherit and modify the destroyed frame's marks. Coercions on the stack are stored as marks and coalesced on tail calls.

7 EARLY ERROR DETECTION

Consider the following code fragment, which erroneously attempts to convert an $(\mathtt{Int} \to \mathtt{Int})$ function to have type $(\mathtt{Bool} \to \mathtt{Int})$:

$$\begin{array}{l}\mathtt{let}\ f :? = (\lambda x : \mathtt{Int}.\ x)\ \mathtt{in} \\ \mathtt{let}\ g : (\mathtt{Bool} \to \mathtt{Int}) = f\ \mathtt{in}\ \ldots\end{array}$$

Prior strategies for gradual typing would not detect this error until g is applied to a boolean argument, causing the integer cast to fail.

In contrast, our coercion-based implementation allows us to detect this error as soon as g is defined. In particular, after cast insertion and evaluation, the value of g is

$$\langle \texttt{Fun Fail } I \rangle \ (\lambda x : \texttt{Int}.\ x)$$

where the domain coercion `Fail` explicates the inconsistency of the two domain types `Int` and `Bool`. We can modify our semantics to halt as soon as such inconsistencies are detected, by adding the following coercion normalization rules:

$$\begin{array}{lcllcl} \texttt{Fun } c \texttt{ Fail} & = & \texttt{Fail} & \texttt{Ref } c \texttt{ Fail} & = & \texttt{Fail} \\ \texttt{Fun Fail } c & = & \texttt{Fail} & \texttt{Ref Fail } c & = & \texttt{Fail} \end{array}$$

Using these rules, our implementation strategy halts as soon as g is defined, resulting in earlier error detection and better test coverage, since g may not actually be called in some test runs.

8 RELATED WORK

There is a large body of work combining static and dynamic typing. The simplest approach is to use reflection with the type ?, as in Amber [3]. Since case dispatch cannot be precisely type-checked with reflection alone, many languages provide statically-typed `typecase` on dynamically-typed values, including Simula-67 [1] and Modula-3 [4].

For dynamically-typed languages, *soft typing* systems provide type-like static analyses for optimization and early error reporting [25]. These systems may provide static type information but do not allow explicit type annotations, whereas enforcing documented program invariants (i.e., types) is a central feature of gradual typing.

Similarly, Henglein's theory of dynamic typing [17] provides a framework for static type optimizations but only in a purely dynamically-typed setting. We use Henglein's coercions instead for structuring the algebra of our cast representation. Our application is essentially different: in the gradually-typed setting, coercions serve to enforce explicit type annotations, whereas in the dynamically-typed setting, coercions represent checks required by primitive operations.

None of these approaches facilitates *migration* between dynamically and statically-typed code, at best requiring hand-coded interfaces between them. The gradually-typed approach, exemplified by Sage [16] and $\lambda^{?}_{\rightarrow}$ [22], lowers the barrier for code migration by allowing mixture of expressions of type ? with more precisely-typed expressions. Our work improves gradual typing by eliminating the drastic effects on space efficiency subtly incurred by crossing the boundary between typing disciplines. Siek and Taha's subsequent work on gradual typing in an object-oriented setting [23] includes evaluation rules to merge some casts at runtime, but not in a sufficiently comprehensive manner to be able to prove bounds on space consumption.

Several other systems employ dynamic function proxies, including hybrid type checking [12], software contracts [11], and recent work on software migration by Tobin-Hochstadt and Felleisen [24]. We believe our approach to coalescing redundant proxies could improve the efficiency of all of these systems.

Minamide and Garrigue [20] address a similar issue with unbounded proxies when optimizing polymorphic functions with monomorphic specializations. They solve the problem by pairing specialized functions with a pointer to their original polymorphic function, so subsequent proxies can always be applied to the original function. They prove the runtime efficiency of their transformation; in our paper we focus on space efficiency.

9 CONCLUSION AND FUTURE WORK

We have presented a space-efficient implementation strategy for the gradually-typed λ-calculus. For simplicity, we have presented our work in the context of a substitution semantics. We would like to extend this work to more realistic models of space consumption for functional languages [21]. Our preliminary results in this direction are promising. More work remains to demonstrate the applicability of this technique in the setting of more advanced type systems. In particular, recursive types and polymorphic types may present a challenge for maintaining constant bounds on the size of coercions. We intend to explore techniques for representing these infinite structures as finite graphs.

Another useful feature for runtime checks is *blame annotations* [11], which pinpoint the particular expressions in the source program that cause coercion failures at runtime by associating coercions with the expressions responsible for them. Blame-tracking improves the error messages for gradually typed programming languages by pinpointing the culprit of failed casts, and also leads to a stronger and more practical type soundness theorem. [24]. It should be possible to track the minimum amount of source location information required for tracking blame, combining space-efficient gradual typing with informative error messages.

ACKNOWLEDGMENTS

David Herman is supported by a grant from the Mozilla Corporation. Aaron Tomb and Cormac Flanagan are supported by NSF grant CCR-0341179.

REFERENCES

[1] G. Birtwhistle et al. *Simula Begin*. Chartwell-Bratt Ltd., 1979.

[2] G. Bracha. Pluggable type systems. In *Workshop on Revival of Dynamic Languages*, October 2004.

[3] L. Cardelli. Amber. In *Spring School of the LITP on Combinators and Functional Programming Languages*, pages 21–47, 1986.

[4] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report (revised). Technical Report 52, DEC SRC, 1989.

[5] C. Chambers. *The Cecil Language Specification and Rationale: Version 3.0*. University of Washington, 1998.

[6] J. Clements. *Portable and high-level access to the stack with Continuation Marks*. PhD thesis, Northeastern University, 2005.

[7] R. B. de Oliveira. The Boo programming language, 2005.

[8] Ecma International. *ECMAScript Language Specification*, 3rd edition, 1999.

[9] Ecma International. *ECMAScript Edition 4 group wiki*, 2007.

[10] R. B. Findler and M. Blume. Contracts as pairs of projections. In *FLOPS*, pages 226–241, 2006.

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

[12] C. Flanagan. Hybrid type checking. In *Symposium on Principles of Programming Languages*, pages 245–256, 2006.

[13] C. Flanagan, S. N. Freund, and A. Tomb. Hybrid types, invariants, and refinements for imperative objects. In *International Workshop on Foundations and Developments of Object-Oriented Languages*, 2006.

[14] S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined style. In *International Conference on Functional Programming*, pages 18–27, 1999.

[15] J. J. Garrett. Ajax: A new approach to web applications, 2005.

[16] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, September 2006.

[17] F. Henglein. Dynamic typing: Syntax and proof theory. *Sci. Comput. Program.*, 22(3):197–230, 1994.

[18] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *POPL '07: Conference record of the 34th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2007.

[19] E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed. In *Workshop on Revival of Dynamic Languages*, 2005.

[20] Y. Minamide and J. Garrigue. On the runtime complexity of type-directed unboxing. In *International Conference on Functional Programming*, pages 1–12, 1998.

[21] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *International Conference on Functional Programming Languages and Computer Architecture*, pages 66–77, 1995.

[22] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.

[23] J. G. Siek and W. Taha. Gradual typing for objects. In *ECOOP 2007: European Conference on Object-Oriented Programming*, Berlin, Germany, July 2007. To appear (Draft posted on TYPES mailing list, 2/12/2007).

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

[25] A. K. Wright. *Practical Soft Typing*. PhD thesis, Rice University, Aug. 1998.