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 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 constraints of a static type system, and without the overhead of documenting (often-changing) structural invariants as types. For large systems, 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 unsupportable. Consequently, several languages attempt to combine static and dynamic typing, among them Cecil [5], Boo [7], Visual Basic.NET [19], and PLT Scheme [22]. This approach of *optional typing* or *gradual typing* has begun to gain attention in the research community [21, 22, 19, 2]. The ability to implement both exploratory, partially-conceived prototypes and mature, production systems in the same programming language offers the possibility of continuous software evolution from prototype to product, thus avoiding the huge costs of language migration. The gradually-typed approach to programming encompasses both the exploratory phase of early software development and its evolution to mature, well-structured production systems.

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[1] 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 gradual typing.

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:

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

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:

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

This program seems innocuous, but suffers from a serious space leak. Since *even* is dynamically typed, the result of each call to $\textit{even}\ (n-1)$ 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, 21], 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:

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

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 $\langle \texttt{Dyn} \to \texttt{Dyn} \rangle$, respectively. This means that the function argument k is wrapped in an additional function proxy at each recursive call!

In summary, existing implementation techniques for gradual typing results in excessive and unnecessary space consumption.

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

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 containes only two distinct components, $\texttt{Bool} \to \texttt{Bool}$ and $\texttt{Dyn} \to \texttt{Dyn}$, which could be merged to the simpler but equivalent $\texttt{Bool} \to \texttt{Bool}$ proxy.

Merging proxies requires a representation of proxies that supports this kind of transformation. Since the representation of proxies as functions is too opaque, our formalization instead leverages Henglein's *coercions* [17], a data structure with algebraic properties that allows us to combine adjacent coercions in order to thus eliminate redundant information and to 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 gradually- or 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 gradually-typed λ-calculus $\lambda^{?}_{\to}$ of Siek and Taha [21].

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. For completeness, we have included details of the semantics and correctness proofs in an appendix, available at:

`http://www.ccs.neu.edu/home/dherman/tfp-appendix.pdf`

The appendix is not required in order to read this extended abstract.

2 GRADUALLY-TYPED LAMBDA CALCULUS

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

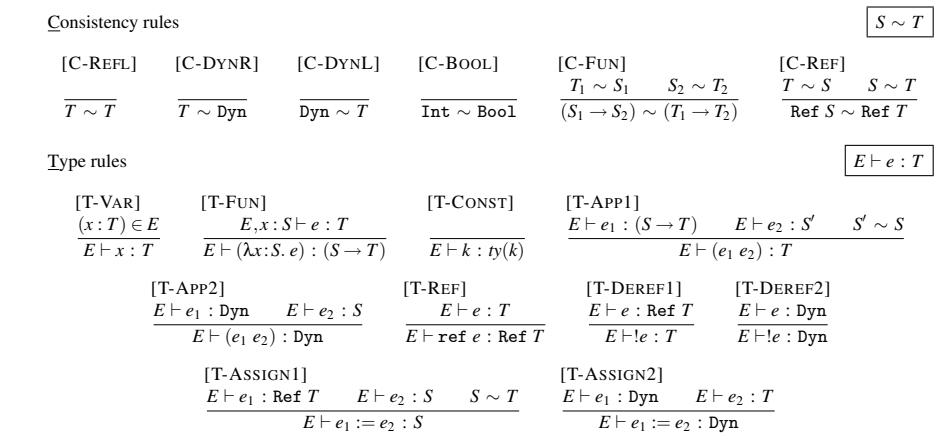
$$\begin{array}{lll} \textit{Terms:} & e & ::= k \mid x \mid \lambda x{:}T.\, e \mid e\ e \mid \texttt{ref}\ e \mid\ !e \mid e := e \\ \textit{Types:} & S,T & ::= B \mid T \to T \mid \texttt{Dyn} \mid \texttt{Ref}\ T \end{array}$$

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

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 `Dyn`, from which it follows that, for example, $\texttt{Bool} \sim \texttt{Dyn}$ and $\texttt{Dyn} \sim \texttt{Int}$. However, booleans cannot be used directly as integers, which is why the consistency relation is not transitively closed. Unlike Siek and Taha, 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 `Dyn`. The rule [C-Bool] serves as an example of asymmetry by

Figure 1: Source Language Type System



allowing coercion from `Int` to `Bool` but not the reverse. The rule [C-FUN] is reminiscent of the contravariant/covariant rule for function subtyping. 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 Dyn` are consistent.

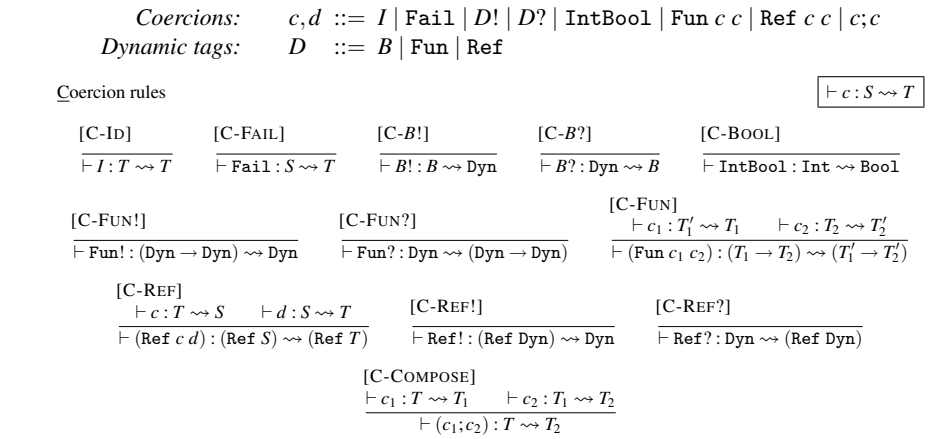
Figure 1 also presents the type rules for the source language, which are mostly standard. Notice the presence of two separate rules for procedure application. Rule [T-APP1] handles the case where the operator is statically-typed as a function; in this case, the argument may have any type consistent with 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 coercions. These coercions are drawn from Henglein's theory of dynamic typing [17], and are used to perform coercions between consistent types. Their key benefit over prior proxy-based implementations 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 `Dyn`, and a corresponding check-and-untag coercion $D?$ that takes values of type `Dyn`. Thus, for example, we have $\vdash$ `Bool!` : `Bool` $\rightsquigarrow$ `Dyn` and $\vdash$ `Bool?` : `Dyn` $\rightsquigarrow$ `Bool`.

Figure 2: Coercion Language And Type Rules



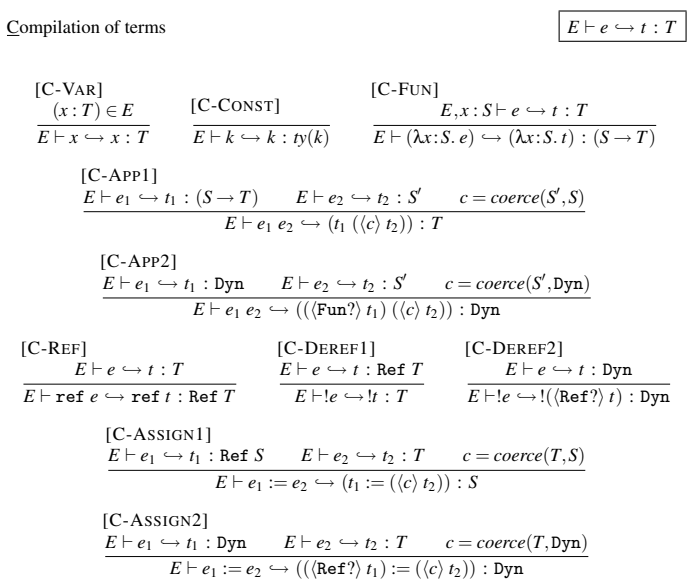
The function checking coercion Fun? converts a value of type Dyn to have the most general function type Dyn → Dyn. If a more precise function type is required, this value can be further coerced via a function coercion Fun c d, where c coerces function arguments and d coerces results. For example, the coercion (Fun Bool? Bool!) coerces from Dyn → Dyn to Bool → Bool, by untagging function arguments (via Bool?) and tagging function results (via Bool!). 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 $coerce(S,T)$ is defined and returns the appropriate coercion between these types.

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

Coercing a type T to itself produces the identity coercion I. Coercing base types B to Dyn requires a tagging coercion B!, and coercing Dyn to a base type B requires a runtime check B?. Function coercions work by coercing their domain and range types. The type Dyn 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 Dyn → Dyn to $T_1 \to T_2$. Dually, typed

Figure 3: Compilation Rules



functions are coerced to type `Dyn` 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 `Dyn` 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 $\vdash c : S \rightsquigarrow T$.*

4 TARGET LANGUAGE AND COMPILATION

During compilation, we both type check the source program and also insert explicit type casts where necessary. The target language of this compilation process is essentially the same as the source, except that it uses explicit coercions of the form $\langle c \rangle\ t$ as the only mechanism for connecting terms of type `Dyn` and terms of other types. For example, the term $\langle \mathtt{Bool?} \rangle\ x$ has type `Bool`, provided that x has type `Dyn`. The language syntax is as follows; its type rules are mostly straightforward (see Figure 4 in the appendix for details).

$$s,t \ ::= \ k \mid x \mid \lambda x{:}T.\, t \mid t\ t \mid \mathtt{ref}\ t \mid\ !t \mid t := t \mid \langle c \rangle\ t$$

The process of type checking and inserting coercions is formalized via the *compilation 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 compilation judgment are shown in Figure 3, and

they rely on the partial function *coerce* to compute coercions between types. For example, rule [C-APP1] compiles an application expression where the operator has a function type $S \to T$ by coercing the argument expression from type S' to S. Compilation succeeds on all well-typed source programs, and produces only well-typed target programs (see appendix for details).

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 coercions, thus eliminating redundant information while preserving the semantic behavior of programs.

Following Morrisett, Felleisen, and Harper [20], we use the CEKS machine [10] as an operational model of the target language that is representative of the space usage of realistic implementations. (Figure 4 of the appendix provides the definitions and rules of the abstract machine.) Each configuration of the machine is either a term configuration $\langle t,\ \rho,\ C,\ \sigma\rangle$ or a value configuration $\langle v,\ C,\ \sigma\rangle$, representing evaluation of a term t in environment ρ or return of a value v, respectively, in an evaluation context C and store σ.

Most of the evaluation rules are standard for a context-machine reduction semantics. The most important rule is [E-CCAST], which ensures that adjacent casts are always merged:

$$\langle\langle c\rangle\ (\langle d\rangle\ t),\ \rho,\ C,\ \sigma\rangle \quad\longrightarrow\quad \langle\langle d;c\rangle\ t,\ \rho,\ C,\ \sigma\rangle \ \text{[E-CCAST]}$$

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

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

This normalization is applied in a transitive, compatible manner whenever the rule [E-CCAST] is applied, thus bounding the size of coercions generated during evaluation. Evaluation satisfies the usual progress and preservation lemmas (see appendix for details).

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

In addition, the height of any coercion created during compilation is bounded by the height of some type in the source program (where the height of a type is the height of its abstract syntax tree representation).

Lemma 3. *If $c = coerce(S,T)$, then $height(c) \leq max(height(S), height(T))$.*

We now bound the total cost of maintaining coercions in the space-efficient semantics. We define the size of a configuration 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:

$$
\begin{aligned}
&size(\langle t,\ \rho,\ C,\ \sigma\rangle) = size(t) + size(\rho) + size(C) + size(\sigma|_{reachable(t,\rho,C,\sigma)}) + 1\\
&size(\rho) = \textstyle\sum_{x\in dom(\rho)}(1 + size(\rho(x)))\\
&size(\langle \lambda x{:}T.\ t, \rho\rangle) = size(\lambda x{:}T.\ t) + size(\rho) + 1\\
&size(k) = size(a) = 1\\
&\ldots
\end{aligned}
$$

To show that coercions occupy bounded space in the model, we compare the size of configurations in reduction sequences to configurations in an "oracle" semantics where coercions require no space. The oracular measure $size_{\text{OR}}$ is defined similarly to *size*, but without a cost for maintaining coercions:

$$
\begin{aligned}
&size_{\text{OR}}(c) = 0\\
&size_{\text{OR}}(\langle c\rangle\ t) = size_{\text{OR}}(t)\\
&size_{\text{OR}}(\langle c\rangle\ \bullet) = 0\\
&\ldots
\end{aligned}
$$

The following theorem then bounds the fraction of the program state occupied by coercions in the space-efficient semantics.

Theorem 4 (Space consumption). *If $\emptyset \vdash e \hookrightarrow t : T$ and $\langle t,\ \emptyset,\ \bullet,\ \emptyset\rangle \longrightarrow^* M$, then there exists some $S \in e$ such that $size(M) \in O(2^{height(S)} \cdot size_{\text{OR}}(M))$.*

Theorem 4 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 immediately combining adjacent coercions on the stack via the [E-CCAST] rule. In our full paper, we sketch three implementation techniques for this rule: *coercion-passing style*, a *trampoline* [14], and *continuation marks* [6].

7 EARLY ERROR DETECTION

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

$$
\begin{aligned}
&\texttt{let}\ f : \texttt{Dyn} = (\lambda x : \texttt{Int}.\ x)\ \texttt{in}\\
&\texttt{let}\ g : (\texttt{Bool} \to \texttt{Int}) = f\ \texttt{in}\ \ldots
\end{aligned}
$$

Prior strategies for gradual typing would not detect this error until g is called.

In contrast, our coercion-based implementation allows us to detect this error as soon as g is defined. In particular, after compilation 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 incompatibility of the two domain types `Int` and `Bool`. We can modify our semantics to halt as soon as such incompatibilities 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 coverage, since g may not actually be called in some tests.

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 `Dyn`, as in Amber [3]. Since case dispatch cannot be precisely type-checked with reflection, 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 that facilitate optimization and early error reporting [23]. 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 do make use of Henglein's coercions for structuring the coercion algebra of our target language. But 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 λ-calculus of Siek and Taha [21], by contrast, lowers the barrier for code migration by allowing mixture of expressions of type `Dyn` with more precisely-typed expressions. Our work expands on the gradually-typed λ-calculus by providing a space-efficient implementation strategy.

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 [22]. We believe our approach to coalescing redundant proxies could improve the efficiency of all of these systems.

9 CONCLUSION AND FUTURE WORK

We have presented a space-efficient implementation strategy for the gradually-typed λ-calculus. 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. It should be possible to track source location information for only the most recent potential culprit for each type of coercion failure, combining space-efficient gradual typing with informative error messages.

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*, third edition, December 1999.

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

[10] M. Felleisen and M. Flatt. Programming languages and lambda calculi. Lecture notes online, July 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] 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.

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

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

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

[24] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.