Toward Efficient Gradual Typing for Structural Types via Coercions

Andre Kuhlenschmidt
Indiana University
USA
akuhlens@indiana.edu

Deyaaeldeen Almahallawi
Indiana University
USA
dalmahal@indiana.edu

Jeremy G. Siek
Indiana University
USA
jsiek@indiana.edu

Abstract

Gradual typing combines static and dynamic typing in the same program. Siek et al. (2015) describe five criteria for gradually typed languages, including type soundness and the gradual guarantee. A significant number of languages have been developed in academia and industry that support some of these criteria (TypeScript, Typed Racket, Safe TypeScript, Transient Reticulated Python, Thorn, etc.), but relatively few support all the criteria (Nom, Gradualtalk, Guarded Reticulated Python). Of those that do, only Nom does so efficiently. The Nom experiment shows that one can achieve efficient gradual typing in languages with only nominal types, but many languages have structural types: function types, tuples, record and object types, generics, etc.

In this paper we present a compiler, named Grift, that addresses the difficult challenge of efficient gradual typing for structural types. The input language includes a selection of difficult features: first-class functions, mutable arrays, and recursive types. We show that a close-to-the-metal implementation of run-time casts inspired by Henglein's coercions eliminates all of the catastrophic slowdowns without introducing significant average-case overhead. As a result, Grift exhibits lower overheads than those of Typed Racket.

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

Keywords gradual typing, compilation, efficiency

ACM Reference Format:
Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. Toward Efficient Gradual Typing for Structural Types via Coercions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/3314221.3314627

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06.
https://doi.org/10.1145/3314221.3314627

1 Introduction

Gradual typing combines static and dynamic type checking, giving the programmer control over which typing discipline to be used in each region of code [5, 30, 50, 61]. In the past decade, considerable progress has been made on the theory of gradual typing, such as understanding interactions with objects [51], generics [3, 4, 37, 38], mutable state [35, 56], recursive and set-theoretic types [14, 53], control operators [46], and type inference [23, 54].

Beginning with the observations of Herman et al. [35, 36], that the standard operational semantics for the Gradual Typed Lambda Calculus can exhibit unbounded space leaks, and continuing with the experiments of Takikawa et al. [59], which showed that Typed Racket [62] exhibits high overheads on real programs, it has become clear that efficiency is a serious concern for gradually typed languages. To address this concern, the research community has made some first steps towards answering the important scientific question: *What is the essential overhead of gradual typing?* This is a difficult and complex question to answer. First, there is a large language design space: choices regarding the semantics of a gradually typed language have significant impacts on efficiency. Second, there is the engineering challenge of developing the implementation technology necessary for performance evaluations. Third, there is the scientific challenge of inventing techniques to improve efficiency. We discuss these three aspects in the following paragraphs and describe where our research program fits in. While we cannot hope to outright answer this question, our paper eliminates some spurious factors and provides a rigorous baseline for further experimentation.

The Language Design Space Siek et al. [55] 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.

For expediency, many languages from industry (TypeScript [9, 33], Hack [63], and Flow [1]) are implemented by erasing types and compiling to untyped languages. This approach does not provide type soundness in the above sense.

Language	Sound	Gradual Guarantee wrt. Structural Types	Gradual Guarantee wrt. Nominal Types	Granularity	Approach	Space-Efficient
Gradualtalk	●	●	●	Fine	Retrofit	○
Guarded Reticulated Python	●	●	●	Fine	Retrofit	○
Nom	●	–	●	Fine	From-Scratch	●
GTLC+	●	●	–	Fine	From-Scratch	●
TypeScript	○	●	●	Fine	Retrofit	●
Safe TypeScript	●	○	○	Fine	Retrofit	●
Typed Racket	●	◐	◐	Coarse	Retrofit	●
Transient Reticulated Python	◐	●	●	Fine	Retrofit	●

Figure 1. A comparison of gradually typed programming languages.

Several of the designs from academia (Thorn [10], TS* [58], Safe TypeScript [43, 44], and Strong Script [45]) place restrictions on which implicit casts are allowed, for the sake of efficiency, but at the price of losing the gradual guarantee. The fundamental tension is that providing both the gradual guarantee and type soundness means that an implementation must perform runtime type checking at the boundaries between statically typed and dynamically typed regions of code, and that runtime checking can be time consuming.

We note that Typed Racket is sound and partially supports the gradual guarantee: its type system does not satisfy the static part of the gradual guarantee because it requires what amounts to an explicit downcast to use a Racket module from a Typed Racket module. However, the semantics of Typed Racket's runtime checks are compatible with the dynamic part of the gradual guarantee.

Another aspect of the language design space that impacts efficiency is whether a gradually typed language includes structural or nominal types. With nominal types, the runtime check for whether a value has a given type is efficient and straightforward to implement. Indeed, Muehlboeck and Tate [42] show that Nom, a nominally-typed object-oriented language (without generics or function types), exhibits low overhead on the `sieve` and `snake` benchmarks from the Gradual Typing Performance Benchmarks [2]. On the other hand, with structural types, the runtime check can be much more complex, e.g., for higher-order types it may involve the use of a proxy to mediate between a value and its context.

Finally, gradual typing can be applied at varying granularities. For example, in Typed Racket, a module may be typed or untyped. We refer to this as *coarse-grained gradual typing*. In contrast, in TypeScript [9, 33] and Reticulated Python [64, 65], each variable may be typed or untyped, and furthermore, a type annotation can be partial through the use of the unknown type. We refer to this as *fine-grained gradual typing*.

In this paper we study the efficiency of gradual typed languages that satisfy the five criteria, that include structural types, and that employ fine-grained gradual typing.

Implementation Technology A popular approach to implementing gradually typed languages is to retrofit a pre-existing language implementation. The benefit of this approach is that it quickly provides support for large number of language features, facilitating performance evaluations on a large number of real programs. The downside is that the pre-existing implementation was not designed for gradual typing and may include choices that interfere with obtaining efficiency on partially typed programs. Many of the gradually typed languages to date have been implemented via compilation to a dynamically typed language, including TypeScript, Typed Racket, Gradualtalk, Reticulated Python, and many more. These implementations incur incidental overhead in the statically typed regions of a program.

The opposite approach is to develop a from-scratch implementation of a gradually typed language. The benefit is that its authors can tailor every aspect of the implementation for efficient gradual typing, but an enormous engineering effort is needed to implement all the language features necessary to run real programs.

For a gradually typed language, one of the most important choices is how to implement runtime type checks. For expediency, Typed Racket uses the Racket contract library [21]. The contract library is more general than is necessary because it supports arbitrary predicates instead of just type tests. In subsequent years since the experiment of Takikawa et al. [59], several performance problems have been fixed, as reported by Bauman et al. [7]. It is unclear how much performance is left on the table given the extra layers of abstraction and indirection in the contract library.

To better isolate the essential overheads of gradual typing, this paper studies a from-scratch implementation in the context of a simple ahead-of-time compiler with a close-to-the-metal implementation of runtime type checks. The alternative of using just-in-time compilation is a fascinating one [7, 44], but we think it is important to also study an implementation whose performance is more predictable, enabling us to more easily isolate the causes of overhead.

Innovations to Improve Efficiency Perhaps the most challenging obstacle to determining the essential overhead of

gradual typing is that, creative researchers continue to make innovations that can lower the overhead of gradual typing! To make claims about the essential overhead of gradual typing, these ideas must be implemented and evaluated.

Herman et al. [35, 36] observed that the *coercions* of Henglein [34] (originally designed for the compile-time optimization of dynamically typed languages) could be used to guarantee space efficiency in the proxies needed for higher-order structural types by normalizing coercions. Space efficiency is guaranteed because arbitrarily long sequence of coercions can always be compressed into an equivalent coercion of length three or less. Thus, the size of a coercion c in normal form is bounded by its height h, $size(c) \leq 5(2^h - 1)$. The height of the coercions generated at compile time and runtime are bounded by the height of the types in the source program. Thus, at any moment during program execution, the amount of space used by the program is $O(n)$, where n is the amount of space ignoring coercions. To date the research on coercions for gradual typing has been of a theoretical nature. The Grift compiler is the first to empirically test the use of coercions to implement runtime casts for gradually typed languages.

For implementations that rely on contracts for runtime checking, space efficiency is also a concern and Greenberg [26] discovered a way to compress sequences of contracts, making them space efficient. Feltey et al. [20] implement this technique, *collapsible contracts*, in the Racket contract library and demonstrate that it significantly improves the performance of Typed Racket on some benchmarks. However, contracts cannot be compressed to the same degree as coercions (predicates are more expressive than types), which means that the time overhead has larger constant factors, factors that depend on the total number of contracts in a program. In our evaluation we compare to a version of Racket uses collapsible contracts.

Another innovation is the notion of *monotonic references* of Siek et al. [56], which has the potential to eliminate the overhead of gradual typing in statically typed regions of code. Richards et al. [44] shows that monotonic references reduce overhead in the context of a JIT implementation of Safe TypeScript. In a forthcoming paper we demonstrate that monotonic references also reduce overhead in the context of the Grift compiler.

In this paper we present evidence that efficiency can be achieved in a fine-grained gradually typed language with structural types. We build and evaluate an ahead-of-time compiler, named Grift, that uses carefully chosen runtime representations to implement coercions.

The input language includes a selection language features that are difficult to implement efficiently in a gradually typed language: first-class functions, mutable arrays, and equirecursive types. The language is an extension of the Gradually Typed Lambda Calculus, abbreviated GTLC+.

Figure 1 summarizes the discussion up to this point. The top-half of the table lists four languages that meet the criteria for gradual typing whereas the bottom-half includes languages that do not provide type soundness and/or the gradual guarantee. Compared to the other three languages that satisfy the criteria, the GTLC+ language (and Grift compiler) described in this paper is novel in its support for structural types and guaranteed space efficiency.

Road Map The paper continues as follows.

- Section 2 provides background on gradual typing.
- Section 3 describes the implementation of the compiler.
- Section 4 presents an empirical performance evaluation of the techniques used in Grift. In particular,
  - Section 4.2 shows that coercions eliminate catastrophic slowdowns without adding significant overhead compared to traditional casts.
  - Section 4.3 evaluates the performance of Grift on typed, untyped, and partially typed benchmarks. To put these results in context we make comparisons to other programming language implementations.

2 Background on Gradual Typing

From a language design perspective, gradual typing touches both the type system and the operational semantics. The key to the type system is the *consistency* relation on types, which enables implicit casts to and from the unknown type, here written `Dyn`, while still catching static type errors [5, 30, 50]. The dynamic semantics for gradual typing is closely related to the semantics of *contracts* [21, 25], *coercions* [34], and *interlanguage migration* [41, 61]. Because of the shared mechanisms with these other lines of research, much of the ongoing research in those areas benefits the theory of gradual typing, and vice versa [15, 17, 18, 27, 28, 31, 40, 57]. In the rest of this section we give a brief introduction to gradual typing and describe the performance challenges.

Consider the classic example of Herman et al. [35] shown in Figure 2. Two mutually recursive functions, `even?` and `odd?`, are written in GTLC+. This example uses continuation passing style to concisely illustrate efficiency challenges in gradual typing. While this example is contrived, the same problems occur in real programs under complex situations [20]. On the left side of the figure we have a partially typed function, named `even?`, that checks if an integer is even. On the right side of the figure we have a fully typed function, named `odd?`, that checks if an integer is odd. With gradual typing, both functions are well typed because implicit casts are allowed to and from `Dyn`. For example, in `even?` the parameter `n` is implicitly cast from `Dyn` to `Int` when it is used as the argument to `=` and `-`. These casts check that the dynamic value is tagged as an integer and perform the conversion needed between the representation of tagged values and integers. Conversely, the value `#t` is cast to `Dyn` because it is used as the argument to a function that expects `Dyn`. This

```
(define even? : (Dyn (Dyn -> Bool) -> Bool)
  (lambda ([n : Dyn] [k : (Dyn -> Bool)])
    (if (= n 0)
        (k #t)
        (odd? (- n 1) k))))

(define odd? : (Int (Bool -> Bool) -> Bool)
  (lambda ([n : Int] [k : (Bool -> Bool)])
    (if (= n 0)
        (k #f)
        (even? (- n 1) k))))
```

Figure 2. Gradually typed even? and odd? functions that have been written in continuation passing style.

cast tags the value with runtime type information so that later uses can be checked. Likewise, in `odd?` the `Int` passed as the first argument to `even?` is cast to `Dyn`.

In addition to casts directly to and from `Dyn`, gradual typing supports casting between types that have no conflicting static type information. Such types are said to be *consistent*. The types of the variables named `k`, `(Dyn -> Bool)` and `(Bool -> Bool)` are consistent with each other. As such, when `k` is passed as an argument to `even?` or `odd?`, there is an implicit cast between these two types. This cast is traditionally implemented by wrapping the function with a proxy that checks the argument and return values [21], but Herman et al. [35] observe that the value of `k` passes through this cast at each iteration, causing a build up of proxies that changes the space complexity from constant to $O(n)$.

In this paper we consider two approaches to the implementation of runtime casts: traditional casts, which we refer to as *type-based casts*, and *coercions*. Type-based casts provide the most straightforward implementation, but the proxies they generate can accumulate and consume an unbounded amount of space as discussed above.

Getting back to the `even?` and `odd?` example, we compare the time and space of type-based casts versus coercions in Figure 4 (left hand side). The three plots show the runtime, number of casts performed, and length of the longest chain of proxies, as the input parameter `n` is increased. The plot concerning longest proxy chains shows that type-based casts accumulate longer chains of proxies as we increase parameter `n`. On the other hand, coercions use a constant amount of space by compressing these proxies chains into a single proxy of constant size.

The appearance of long proxy chains can also change the time complexity of a program. We refer to such a change as a *catastrophic slowdown*. Figure 3 shows the code for the quicksort algorithm. The program is statically typed except for the the vector parameter of `sort!`. The single `Dyn` annotation in this type causes runtime overhead inside the auxiliary

```
(define sort! : ((Vect Int) Int Int -> ())
  (lambda ([v : (Vect Dyn)]
           [lo : Int] [hi : Int])
    (when (< lo hi)
      (let ([pivot : Int (partition! v lo hi)])
        (sort! v lo (- pivot 1))
        (sort! v (+ pivot 1) hi)))))

(define swap! : ((Vect Int) Int Int -> ())
  (lambda ([v : (Vect Int)] [i : Int] [j : Int])
    (let ([tmp : Int (vector-ref v i)])
      (vector-set! v i (vector-ref v j))
      (vector-set! v j tmp1))))

(define partition! : ((Vect Int) Int Int -> Int)
  (lambda ([v : (Vect Int)] [l : Int] [h : Int])
    (let ([p : Int (vector-ref v h)]
          [i : (Ref Int) (box (- h 1))])
      (repeat (j l h)
        (when (<= (vector-ref v j) p)
          (box-set! i (+ (unbox i) 1))
          (swap! v (unbox i) j)))
      (swap! v (+ (unbox i) 1) h)
      (+ (unbox i) 1))))
```

Figure 3. The `sort!` function implements the Quicksort algorithm in the GTLC+.

`partition!` and `swap!` functions. Like function types, reference types require proxying that apply casts during read and write operations. Again, a naive implementation of casts allows the proxies to accumulate; each recursive call to `sort!` causes a cast that adds a proxy to the vector being sorted. In quicksort, this changes the worst-case time complexity from $O(n^2)$ to $O(n^3)$ because each read (`vector-ref`) and write (`vector-set!`) traverses a chain of proxies of length $O(n)$.

Returning to Figure 4, but focusing on the right-hand side plots for quicksort, we observe that, for typed-based casts, the longest proxy chain grows as we increase the size of the array being sorted. On the other hand, coercions do extra work at each step to compress the cast. As a result they pay more overhead for each cast, but when they use the casted value later the overhead is guaranteed to be constant. This can be seen in the way the runtime grows rapidly for type-based casts, while coercions remain (relatively) low. We confirmed via polynomial regression that the type-based cast implementation's runtime is modeled by a third degree polynomial, i.e. $O(n^3)$.

Review of Coercions Coercions are combinators that specify how to convert from one type to another type. The following grammar shows the coercions needed to represent casts between types that include the unknown type `Dyn`, base

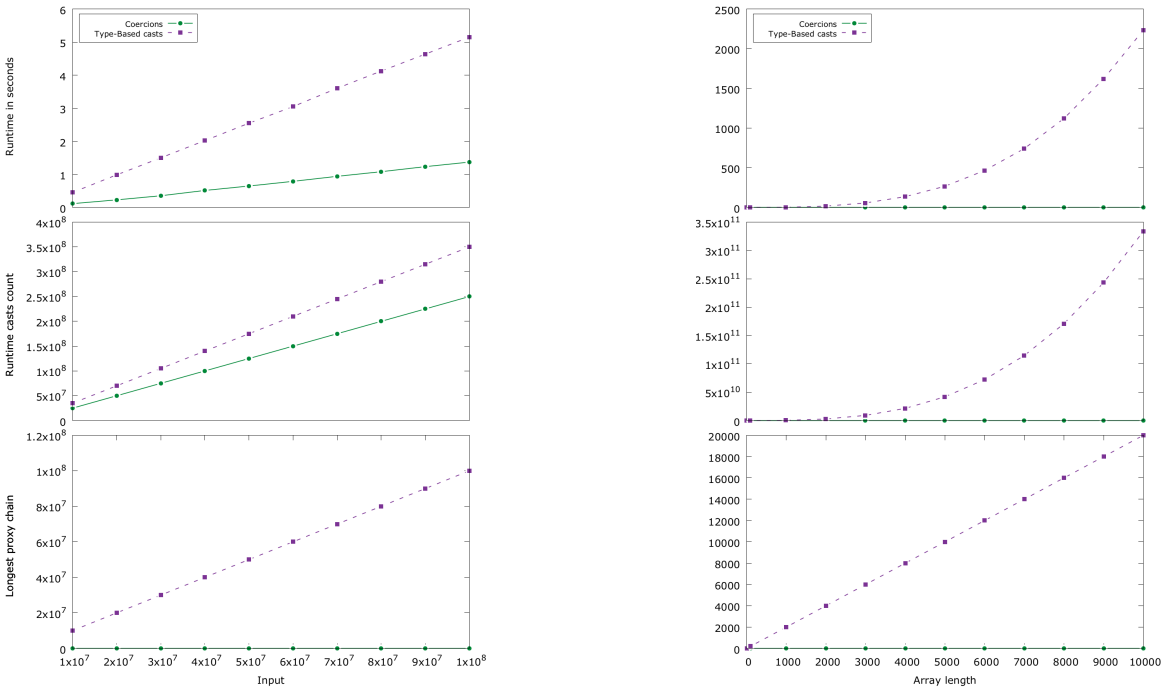


Figure 4. The runtime, number of casts, and longest proxy chain (y-axes) as we increase the parameter n (x-axis) for even/odd (left) and the array length (x-axis) for quicksort (right). The plots for longest proxy chain show that coercions compress casts and thus operate in constant space. The plot of runtime for quicksort shows that long proxy chains can also increase the asymptotic time complexity of an algorithm.

types (integers and Booleans), function types, products, and mutable references.

$$\begin{array}{lll} T & ::= & \mathsf{Dyn} \mid \mathsf{Int} \mid \mathsf{Bool} \mid T \to T \mid T \times T \mid \mathsf{Ref}\ T \\ c, d & ::= & T?^p \mid T! \mid \iota \mid c\ ;\ d \mid \bot^p \mid c \to d \mid c \times d \mid \mathsf{Ref}\ c\ d \end{array}$$

The coercion $T?^p$ is a projection that checks whether a tagged value is of type T. If it is, the underlying value is returned. If not, an error is signaled at the source location p (a blame label). The coercion $T!$ in an injection that tags a value with its type. The identity coercion ι just returns the input value. The sequence $c\ ;\ d$ applies the coercion c then d. The failure coercion $\bot^p$ signals an error when applied to a value. A function coercion $c \to d$ changes the type of a function by applying c to the argument and d to the return value. A product coercion $c \times d$ changes the type of a pair by applying c to the first element and d to the second. A reference coercion $\mathsf{Ref}\ c\ d$ changes the type of a mutable reference by applying c when reading and d when writing.

Siek et al. [52] define a recursive composition operator, written $c \fatsemi d$, that takes two coercions in normal form and directly computes the normal form of sequencing them together. The Grift compiler implements this approach using efficient bit-level representations.

3 The Grift Compiler

Grift compiles the GTLC+ to C, then uses the Clang compiler to generate x86 executables. The Clang compiler provides low level optimizations. The GTLC+ language includes base types such as integers (fixnums), double precision floats, and Booleans but does not support implicit conversions between base types (i.e. no numeric tower). The GTLC+ also includes structural types such as n-ary tuples, mutable vectors, and higher-order functions. Figure 5 defines the syntax of the GTLC+; the operational semantics is defined in Appendix B. Grift does not yet implement space-efficient tail recursion, but Herman et al. [35] and Siek and Garcia [48] describe implementation strategies for doing so. This section presents a high level description of the techniques used to generate code for coercions. The code for Grift is available at the URL https://github.com/Gradual-Typing/Grift/tree/pldi19.

The first step in the Grift compiler is to translate to an intermediate language with explicit casts. This process is standard [36, 47, 50] except that we add a local optimization to avoid unnecessary casts in untyped code. The standard cast insertion algorithm [55] can cause unnecessary overhead in untyped regions of code. Consider the function `(lambda ([f : Dyn]) (f 42))` which applies a dynamically typed value `f` as a function. The standard algorithm would compile it to:

Variables	x	::=	(lisp style identifiers)
Characters	c	::=	(lisp style character literals)
Integers	i	::=	(signed 61 bit integers)
Floats	f	::=	(double precision floating point numbers)
Blame Label	l	::=	(double quoted strings)
Types	T	::=	x | `Dyn` | `Unit` | `Bool` | `Int` | `Char` | `Float` | ($T \ldots$ `->` T) | (`Tuple` $T \ldots$) | (`Ref` T)|(`Vect` T)|(`Rec` x T)
Literals	V	::=	() | `#f` | `#t` | c | i | f
Operators	O	::=	`+` | `-` | `*` | `/` | `<` | `<=` | `=` | `>=` | `>` | `fl+` | `fl-` | `fl*` | `fl/` | `fl<` | `fl<=` | `fl=` | `fl>=` | `fl>`
		|	`int->char` | `char->int` | `float->int` | `int->float`
		|	`print-int` | `read-int` | `print-float` | `print-char` | `read-char`
Parameters	F	::=	x | (x : T)
Expressions	E	::=	V | (O $E \ldots$) | (`ann` E T l) | (`if` E E E) | (`time` E) | x | (`lambda` ($F \ldots$) : T E) | (E $E \ldots$)
		|	(`let` ([x : T E] $\ldots$) $E \ldots$) | (`letrec` ([x : T E] $\ldots$) $E \ldots$) | (`tuple` $E \ldots$) | (`tuple-proj` E i)
		|	(`repeat` (x E E) [(x E)] E) | (`begin` $E \ldots E$) | (`box` E) | (`unbox` E) | (`box-set!` E E)
		|	(`make-vector` E E) | (`vector-ref` E E) | (`vector-set!` E E E) | (`vector-length` E)
Definitions	D	::=	(`define` x : T E) | (`define` (x $F \ldots$) : T $E \ldots$) | E
Program	P	::=	$D \ldots$

Figure 5. The syntax of the GTLC+ as supported by Grift. This grammar shows every major syntactic form available in GTLC+, and presents a handful of the operators. Most type annotations can be omitted by dropping the preceding ":". The syntax for the omitting type annotations in the exception, formal parameters, is shown in the grammar.

```
(lambda ([f : Dyn])
  ((cast f Dyn (Dyn -> Dyn) L) (cast 42 Int Dyn L)))
```

The cast on `f` will allocate a function proxy if the source type of `f` is anything but `(Dyn -> Dyn)`. Although the proxy is important to obtain the desired semantics, the allocation is unnecessary in this case because the proxy is used right away and never used again. Instead, Grift specializes these cases by generating code that does what a proxy would do without actually allocating one. Grift applies this optimization to proxied references and tuples as well.

The next step in the compiler exposes the runtime functions that implement casts. We describe the representation of values in 3.1. We describe the implementation of coercions in Section 3.2. After lowering casts, Grift performs closure conversion using a flat representation [6, 12, 13], and translates all constructors and destructors to memory allocations, reads, writes, and numeric manipulation.

For memory allocation and reclamation, Grift uses the Boehm-Demers-Weiser conservative garbage collector [11, 16]. Grift optimizes closures, for example, translating some closure applications into direct function calls using the techniques of Keep et al. [39]. Grift does not perform any other general-purpose or global optimizations, such as type inference and specialization, constant folding, copy propagation, or inlining. On the other hand, the compiler does specialize casts based on their source and target type and it specializes operations on proxies.

3.1 Value Representation

Values are represented according to their type. An Int value is a 61-bit integer stored in 64-bits. A Bool value is also stored in 64-bits, using the C encoding of 1 for true and 0 for false. A function value is a pointer to one of two different kinds of closures; the lowest bit of the pointer indicates which kind. The first kind, for regular functions, is a flat closure that consists of 1) a function pointer, 2) a pointer to a function for casting the closure, and 3) the values of the free variables. The second kind of closure, which we call a *proxy closure*, is for functions that have been cast. It consists of 1) a function pointer (to a "wrapper" function), 2) a pointer to the underlying closure, and 3) a pointer to a coercion.

A value of reference type is a pointer to the data or to a proxy. The lowest bit of the pointer indicates which. A proxy consists of a reference and a pointer to a reference coercion. A value of type Dyn is a 64-bit integer, with the 3 lowest bits indicating the type of the value that has been *injected* (i.e. cast into the Dyn type). For types whose values can fit in 61 bits (e.g. Int and Bool), the injected value is stored inline. For types with larger values, the 61 bits are a pointer to a pair of 64-bit items that contain the injected value and its type. In the following section, the macros for allocating and accessing values have all uppercase names to distinguish them from C functions. The macro definitions are listed in Appendix A.

3.2 Implementation of Coercions

Coercions are represented by heap allocated values. In Grift, the coercions that are statically known are allocated once at the start of the program. The runtime function `coerce`, described below, implements coercion application. To do so, it interprets the coercion and performs the actions it represents to the value.

Coercions are represented as 64-bit values where the lowest 3 bits indicate whether the coercion is a projection, injection, sequence, failure, or identity. For an identity coercion, the remaining 61 bits are not used. For the other coercions, the 61 bits store a pointer to heap-allocated structures that we describe below. Because the number of pointer tags is limited, the function, reference, tuple, and recursive coercions are differentiated by a secondary tag stored in the first word of their heap-allocated structure. The C type definitions for coercions are included in Appendix A.

- Projection coercions ($T?^p$) cast from Dyn to a type T. The runtime representation is 2×64 bits: the first word is a pointer to the type T and the second is a pointer to the blame label p.
- Injection coercions ($T!$) cast from an arbitrary type to Dyn. The runtime representation is 64 bits, holding a pointer to the type T.
- Function coercions ($c_1, \ldots, c_n \to d$) cast between two function types of the same arity. A coercion for a function with n parameters is represented in $64\times(n+2)$ bits, where the first word stores the secondary tag and arity, the second stores a coercion on the return, and the remaining words store n coercions for the arguments.
- Reference coercions (Ref c d) cast between box types or vector types and are represented as 3×64 bits, including the secondary tag, a coercion for writing, and another coercion for reading.
- Tuple coercions cast between two n-tuple types and are represented as $64\times(n+1)$ bits, including the secondary tag, the length of the tuple, and a coercion for each element of the tuple.
- Recursive coercions (Rec $x.c$) serve as targets for back edges in "infinite" coercions created by casting between equirecursive types. They are represented in 2×64 bits for a secondary tag and a pointer to a coercion whose subcoercions can contain a pointer to this coercion.
- Sequences coercions ($c\,;\,d$) apply coercion c then coercion d and store two coercions in 2×64 bits.
- Failure coercions ($\bot^p$) immediately halt the program and are represented in 64 bits to store a pointer to the blame label.

Applying a Coercion The application of a coercion to a value is implemented by a C function named `coerce`, shown in Figure 6, that takes a value and a coercion and either returns a value or signals an error. The `coerce` functions dispatches on the coercion's tag. Identity coercions return the value unchanged. Sequence coercions apply the first coercion and then the second coercion. Injection coercions build a value of type Dyn. Projection coercions take a value of type Dyn and build a new coercion from the runtime type to the target of the projection, which it applies to the underlying value.

```
obj coerce(obj v, crcn c) {
  switch(TAG(c)) {
  case ID_TAG: return v;
  case SEQUENCE_TAG:
    sequence seq = UNTAG_SEQ(c);
    return coerce(coerce(v, seq->fst), seq->snd);
  case PROJECT_TAG:
    projection proj = UNTAG_PRJ(c);
    crcn c2 = mk_crcn(TYPE(v), proj->type, proj->lbl);
    return coerce(UNTAG(v), c2);
  case INJECT_TAG:
    injection inj = UNTAG_INJECT(c);
    return INJECT(v, inj->type);
  case HAS_2ND_TAG: {
    switch (UNTAG_2ND(c)->second_tag) {
    case REF_COERCION_TAG:
      if (TAG(v) != REF_PROXY) {
        return MK_REF_PROXY(v, c);
      } else {
        ref_proxy p = UNTAG_REF_PROXY(v);
        crcn c2 = compose(p->coercion, c);
        return MK_REF_PROXY(p->ref, c2); }
    case FUN_COERCION_TAG:
      if (TAG(v) != FUN_PROXY) {
        return UNTAG_FUN(v).caster(v, c);
      } else {
        fun_proxy p = UNTAG_FUN_PROXY(v);
        crcn c2 = compose(p->coercion, c);
        return MK_FUN_PROXY(p->wrap, p->clos, c2); }
    case TUPLE_COERCION_TAG:
      int n = TUPLE_COERCION_SIZE(c);
      obj t = MK_TUPLE(n);
      for (int i = 0; i < n; i++) {
        obj  e = t.tup->elem[i];
        crcn d = TUPLE_COERCION_ELEM(c, i);
        t.tup->elem[i] = coerce(e, d); }
      return t;
    case REC_COERCION_TAG:
      return coerce(v, REC_COERCION_BODY(c)); }}
  case FAIL_TAG: raise_blame(UNTAG_FAIL(c)->lbl); }}
```

Figure 6. The `coerce` function applies a coercion to a value.

Coercing a reference type (i.e. box or vector) builds a proxy that stores two coercions, one for reading and one for writing, and the pointer to the underlying reference. In case the reference has already been coerced, the old and new coercions are composed via `compose`, so that there will only ever be one proxy on a proxied reference, which ensures space efficiency.

When coercing a function, `coerce` checks whether the function has previously been coerced. If it has not been previously coerced, then there is no proxy, and we invoke its function pointer for casting, passing it the function and the coercion to be applied. This "caster" function allocates and initializes a proxy closure. If the function has been coerced,

Grift builds a new proxy closure containing the underlying closure, but its coercion is the result of composing the proxy's coercion with the coercion being applied via `compose` (the code for this function is in Appendix A). The call to `compose` is what maintains space efficiency.

Coercing a tuple allocates a new tuple whose elements are the result of recurring on each of elements of the original tuple with each of the corresponding subcoercions of the original tuple coercion. A recursive coercion is simply a target for specifying a recursive coercion. As such, we ignore it and applied the body of the recursive coercion to value. Failure coercions halt execution and report an error.

Applying Functions Because the coercion implementation distinguishes between regular closures and proxy closures, one might expect closure call sites to branch on the type of closure being applied. However, this is not the case because Grift ensures that the memory layout of a proxy closure is compatible with the regular closure calling convention. The only change to the calling convention of functions is that we have to clear the lowest bit of the pointer to the closure which distinguishes proxy closures from regular closures. This representation is inspired by a technique used in Siek and Garcia [48] which itself is inspired by Findler and Felleisen [22].

Reading and Writing to Proxied References To handle reads and writes on proxied references, Grift generates code that branches on whether the reference is proxied or not (by checking the tag bits on the pointer). If the reference is proxied the read or write coercion from the proxy is applied to the value read from or written to the reference. To ensure space efficiency, there can be at most one proxy on each reference. If the reference isn't proxied, the operation is a simple machine read or write.

4 Performance Evaluation

In this performance evaluation, we seek to answer a number of research questions regarding the runtime overheads associated with gradual typing.

1. **What is the time cost of achieving space efficiency with coercions?** (Section 4.2)
2. **What is the overhead of gradual typing?** (Sec. 4.3) We subdivide this question into the overheads on programs that are (a) statically typed, (b) untyped, and (c) partially typed.

Of course, to answer research question 2 definitively we would need to consider all possible implementations of gradual typing. Instead, we only answer this question for the concrete implementation Grift.

4.1 Experimental Methodology

We use benchmarks from a number of sources: the well-known Scheme benchmark suite (R6RS) used to evaluate the Larceny [32] and Gambit [19] compilers, the PARSEC benchmarks [8], the Computer Language Benchmarks Game (CLBG), and the Gradual Typing Performance Benchmarks [2]. We do not use all of the benchmarks from these suites due to the limited number of language features currently supported by the Grift compiler. We continue to add benchmarks as Grift grows to support more language features. In addition to the above benchmarks, we also include two textbook algorithms: matrix multiplication and quicksort. We chose quicksort in particular because it exhibits catastrophic overheads. The benchmarks that we use are:

sieve (GTP) This program finds prime numbers using the Sieve of Eratosthenes. The program includes a library for streams implemented using higher-order functions and the code for sieve itself. We adapt the source for both Typed-Racket and GTLC+ to use equirecursive types instead of nominal types for streams.

n-body (CLBG) Models the orbits of Jovian planets, using a simple symplectic-integrator.

tak (R6RS) This benchmark, originally a Gabriel benchmark, is a triply recursive integer function related to the Takeuchi function. It performs the call `(tak 40 20 12)`. It is a test of function calls and arithmetic.

ray (R6RS) Ray tracing a scene, 20 iterations. It is a test of floating point arithmetic adapted from Graham [24].

blackscholes (PARSEC) This benchmark, originally an Intel RMS benchmark, calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation. There is no closed-form expression for the Black-Scholes equation and as such it must be computed numerically.

matmult (textbook) A triply-nested loop for matrix multiplication, with integer elements. The matrix size is 400×400.

quicksort (textbook) The quicksort algorithm on already-sorted (worst-case) input, with integer arrays of size $10,000$ in the comparison to other languages and $1,000$ for the partially typed programs.

fft (R6RS) Fast Fourier Transform on $1,048,576$ real-valued points in the comparison to other languages and $65,536$ for the partially typed program. A test of floating point numbers.

Porting the Benchmarks. We ported the benchmarks to the GTLC+, OCaml, Typed Racket, Chez Scheme, and Gambit. For the R6RS and CLBG benchmarks, we added types and converted tail recursive loops to an iterative form. For the Blackscholes benchmark, we use the GTLC+ types which are the closest analog to the representation used in the original C benchmark. In some cases the choice of representation in GTLC+ has a more specialized representation than in its original source language, in these cases we alter the original benchmark to make the comparison as close as possible. For Chez Scheme and Gambit we use the safe variants of fixnum

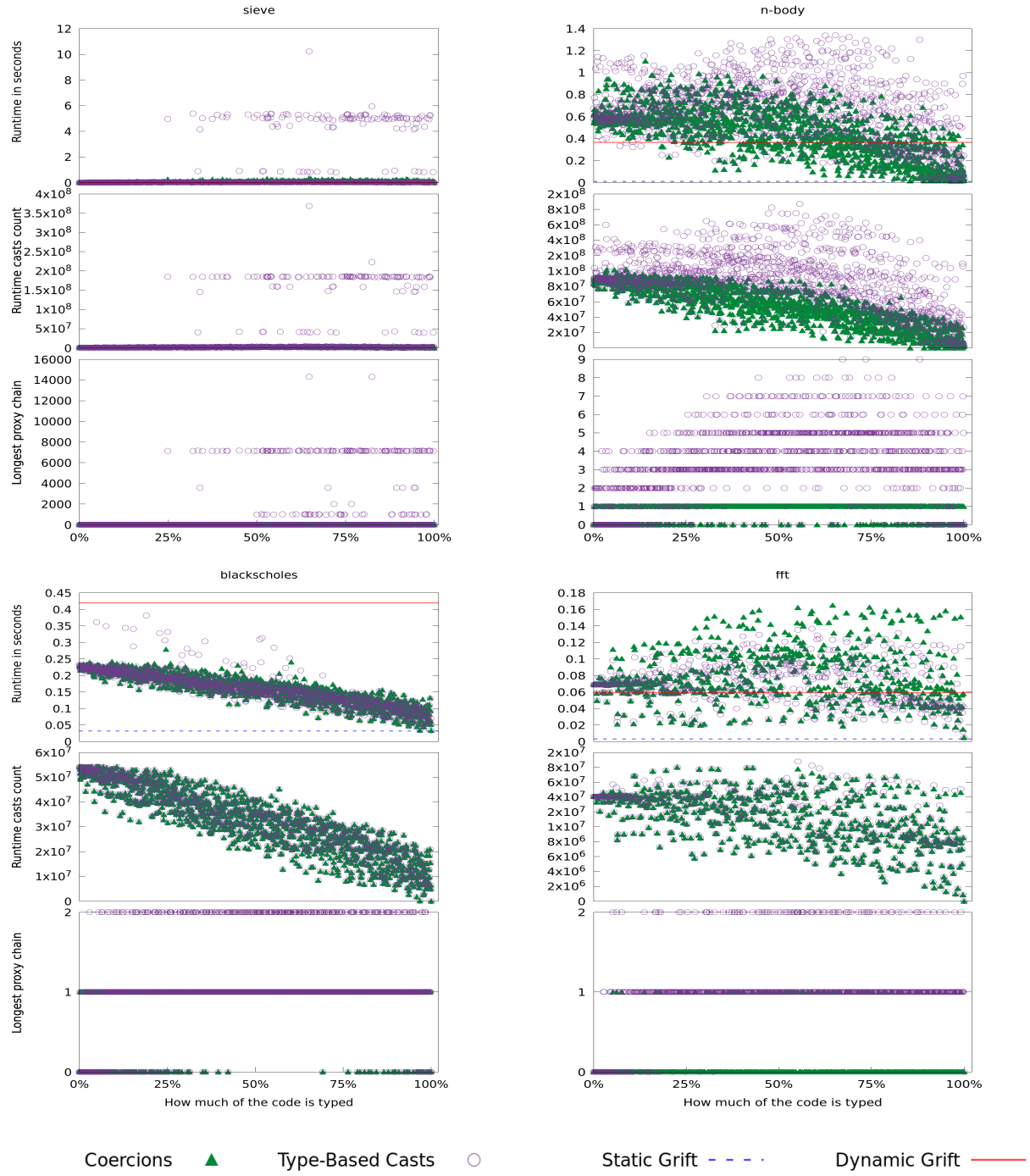


Figure 7. We compare Grift with coercions to Grift with type-based casts across partially typed-programs to evaluate the cost of space-efficiency.

and floating point specialized math operations, but for Racket and Typed-Racket there is only the option of safe and well-performing floating point operators. For fixnums we are forced to use the polymorphic math operations. In OCaml, we use the `int` and `float` types which correspond to unboxed 63 bit integers and boxed double precision floating point numbers respectively. In all languages we use internal timing so that any differences in runtime initialization do not count towards runtime measurements. We make no attempt to account for the difference in garbage collection between the languages. We note that Grift uses an off the shelf version of the

Boehm-Demers-Weiser conservative garbage collector that implements a generational mark-sweep algorithm [11, 16]. The source code for all benchmarks is available at URL: https://github.com/Gradual-Typing/benchmarks/tree/pldi19.

Experimental Setup. The experiments were conducted on an unloaded machine with a 4-core Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz processor with 8192 KB of cache and 16 GB of RAM running Red Hat 4.8.5-16. The C compiler was Clang 7.0.1, the Gambit compiler is version 4.9.3, Racket is version 7.2, and Chez Scheme is version 9.5.3. All time measurements use real time and we report the mean of 5 repeated measurements.

Measuring the Performance Lattice. Takikawa et al. [59] observe that evaluating the performance of implementations of gradually typed languages is challenging because one needs to consider not just one version of each program, but the many versions of a program that can be obtained by adding/removing type annotations. For languages with coarse-grained gradual typing, one considers all the combinations of making each module typed or untyped, so there are 2^m configurations of the m modules. The situation for languages with fine-grained gradual typing, as is the case for GTLC+, is considerably more difficult because any type annotation, and even any node within a type annotation, may be changed to `Dyn`, so there are millions of ways to add type annotations to these benchmarks.

Greenman and Migeed [29] provide evidence that sampling even a linear number of configurations (with respect to program size) gives an accurate characterization of the performance of the exponential configuration space. For our experiments on partially typed programs, we follow the same approach and show the results for a linear number of randomly sampled configurations for each benchmark. We have data for many more configurations, but it does not provide new information and makes the scatter plots more difficult to read.

Our sampling algorithm takes as inputs a statically-typed program, the number of samples, and the number of bins to be uniformly sampled. It creates a list of associations between source locations and type annotations, and shuffles the list to ensure randomness. The algorithm then proceeds to pick new gradual versions of each static type, but constrains the overall program's type precision to fall within a desired bin. These newly generated gradual types are then use to generate a gradual version of the original program by inserting the gradual types at the source locations where the static types where originally found. The algorithm iterates selecting an equal number of samples for each bin until the desired number of samples have been generated.

4.2 The Runtime Cost of Space Efficiency

In Figure 7 we compare the performance of Grift with type-based casts to Grift with coercions, to measure the runtime cost (or savings) of using coercions to obtain space efficiency. We compare these two approaches on partially typed configurations of the benchmarks. We chose the quicksort, n-body, blackscholes, and fft benchmarks as representative examples of the range in performance. The results for the rest of the benchmarks are in Appendix C.

In Figure 7, for each benchmark there are three plots that share the same x-axis, which varies the amount of type annotations in the program, from 0% on the left to 100% on the right. In the first plot, the y-axis is the absolute runtime in seconds. In the second, it is the number of runtime casts that were executed, and in the third plot, the y-axis it is the length of the longest chain of proxies that was accessed at runtime, like the plots regarding even/odd and quicksort in Figure 4. The line marked Dynamic Grift indicates the performance of Grift (using coercions) on untyped code. That is, on code in which every type annotation is Dyn and every constructed value (e.g. integer constant) is explicitly cast to Dyn. The line marked Static Grift is the performance of the Static Grift compiler on fully typed code. Static Grift is a variant of Grift that is statically typed, with no support for (or overhead from) gradual typing (see Section 4.3).

The sieve benchmark elicits very long chains of proxies on some configurations, which in turn causes catastrophic overhead for type-based casts. Indeed, the plot concerning the longest proxy chains for sieve in Figure 7, shows that the configurations with catastrophic performance are the ones that accumulate proxy chains of length greater than 2000. Likewise, Takikawa et al. [59] reported overheads of over 100× on sieve for Typed Racket. In contrast, the coercion-based approach successfully eliminates these catastrophic slowdowns in sieve. The scale of the figure makes it hard to judge their performance in detail as the runtimes are so low relative to the type-based casts. Coercions are 0.32× to 87× faster than type-based casts on sieve.

The n-body benchmark is interesting in that it elicits only mild space efficiency problems, with proxy chains up to length 9, and this corresponds to a mild increase in performance of coercions relative to type-based casts. Coercions are 0.38× to 39× faster than type-based casts on n-body.

In benchmarks that do not elicit space efficiency problems, we see a general trend that coercions are roughly equal in performance to type-based casts.

> **Answer to research question (1):** On benchmarks that do not induce long proxy chains, we sometimes see a mild speedup and sometimes a mild slowdown for coercions compared to type-based casts. However, on benchmarks with long proxy chains, coercions eliminate the catastrophic overheads.

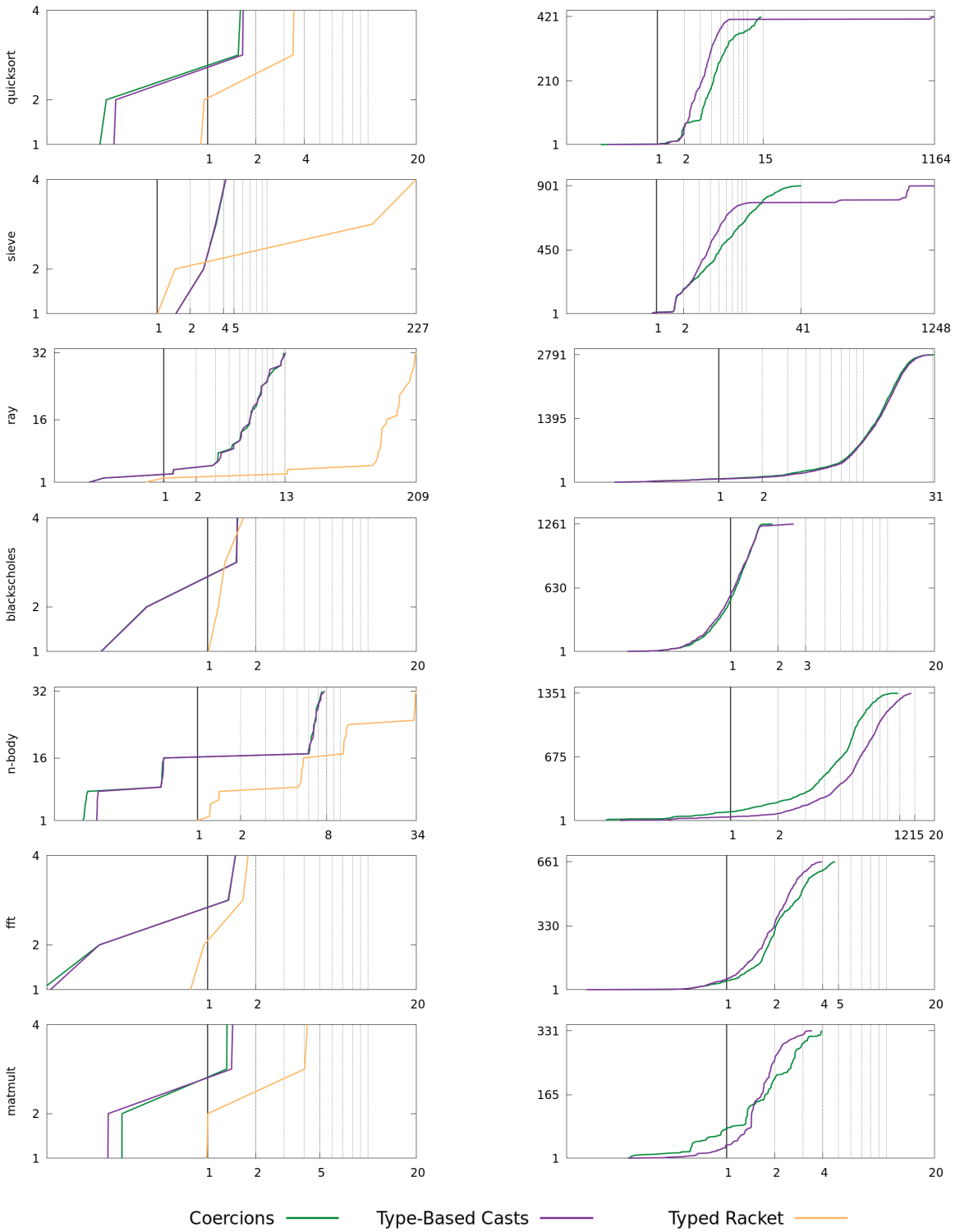


Figure 8. The cumulative performance of Grift and Typed Racket on partially typed configurations. The x-axis represents the slowdowns with respect to Racket. The y-axis is the total number of configurations. The plots on the left are for coarse-grained configurations whereas the plots on the right are for fine-grained configurations (so it is a different view on the same data as in Figure 7). These results show that coercions eliminate the catastrophic overheads (quicksort, sieve) of type-based casts and that Grift has less incidental overhead than Typed Racket (sieve, ray, n-body).

4.3 Gradual Typing Overhead and Comparison

The purpose of this section is to answer research question 2, i.e., what is the overhead of gradual typing? We want to understand which overheads are *inherent* (an necessary part of sound gradual typing) as opposed to *incidental* (unnecessary overheads that could be removed). We identify *incidental* overheads by comparing Grift to other implementations, and reason about what these comparisons say about gradual tying as a whole, and our implementations of Gradual Typing.

In Section 4.3.1 we compare Grift to statically typed programming languages. To isolate the overheads associated with gradual typing from the rest of our implementation, we add a variant of the Grift compiler, named Static Grift, that requires the input program to be statically typed and does not generate any code in support of gradual typing. Comparing Grift to Static Grift allows us to see that gradual typing introduces some overhead (though it doesn't tell us if it is inherent to gradual typing), and comparing Static Grift to OCaml and Typed Racket shows that Static Grift has reasonable performance for a statically typed language.

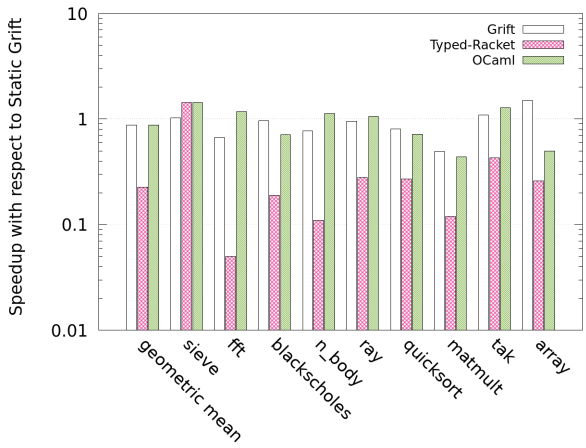
In Section 4.3.2 we examine overheads of gradual typing on dynamically typed code. We compare against Racket, Gambit, and Chez Scheme and find that while Grift is in the ball park of dynamically typed programming languages, it experiences some incidental overheads. We know this because Typed Racket avoids similar overheads, in their dynamically typed implementation. This is as expected because Grift does not implement the many general purpose optimizations that are in these other systems.

In Section 4.3.3 we inspect the overheads of gradual typing on partially typed code. Grift shows that space-efficient coercions avoid the catastrophic overheads associated with gradual typing. This demonstrates that these catastrophic overheads are incidental to gradual typing. On the other hand, we conjecture that constant-factor overheads associated with composing coercions is inherent for gradually typed programming languages with structural types. However, we also think there is still some remaining constant-factor overhead that is incidental and could be eliminated.

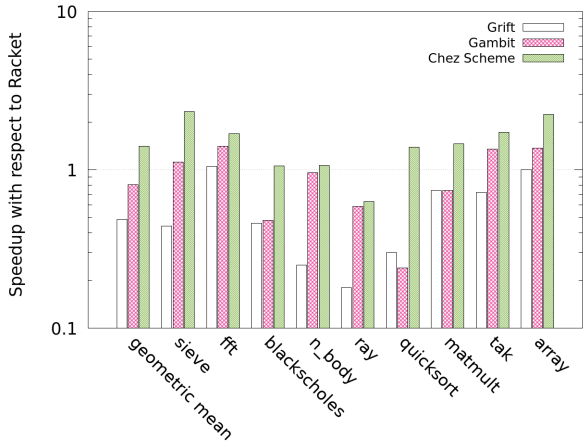
4.3.1 Evaluation on Statically Typed Programs

Figure 9a shows the results of evaluating the speedup of Grift with respect to Static Grift on statically typed versions of the benchmarks. The performance of Grift sometimes dips to 0.49× that of Static Grift. To put the performance of Grift in context, it is comparable to OCaml and better than fully static Typed Racket.

> **Answer to research question (2 a):** the performance of Grift on statically typed code is often reasonable and is on par with OCaml but can dip to 0.49× with respect to Static Grift on array-intensive benchmarks.



(a) Statically Typed Programs



(b) Untyped Programs

Figure 9. A comparison of the speedup on typed and untyped programs of Grift. For typed programs, we measure speedup wrt. Static Grift and compare Grift with OCaml and Typed Racket. Grift shows some overhead compared to Static Grift but is on par with OCaml and significantly faster than Typed Racket. For untyped programs, we measure speedup wrt. Racket and compare to Gambit and Chez Scheme. Grift's performance is roughly half that of Racket, Gambit, and Chez Scheme's.

We believe that most of the differences between Grift and Static Grift can be attributed to the checks for proxies in operations on mutable arrays. We conjecture that this performance overhead is inherent to the standard semantics for gradual typing.

4.3.2 Evaluation on Untyped Programs

Figure 9b shows the results of evaluating the speedup of Grift with respect to Racket on untyped configurations of the benchmarks. The figure also includes results for Gambit

and Chez Scheme. We see that the performance of Grift is generally lower than Racket, Gambit, and Chez Scheme on these benchmarks, which is unsurprising because Grift does not perform any general-purpose optimizations. This experiment does not tease apart which of these performance differences are due to gradual typing per se and which of them are due to orthogonal differences in implementation, e.g., ahead-of-time versus JIT compilation, quality of general-purpose optimizations, etc. Thus we can draw only the following conservative conclusion.

> **Answer to research question (2 b)** the overhead of Grift on untyped code is currently reasonable but there are still some constant-factor improvements to be made.

4.3.3 Evaluation on Partially Typed Programs

To answer research question (2 c), i.e., "what is the overhead of gradual typing for partially typed code?", we consider the results in Figure 8. The left-hand column shows the performance of Grift (with coercions and type-based) and for Typed Racket on coarse-grained configurations, in which each module is either typed or untyped. The right-hand column shows the performance results for Grift on fine-grained configurations.

The cumulative performance plots shown in Figure 8 indicate how many partially typed configurations perform within a certain performance range. The x-axis is log-scale slowdown with respect to Racket and the y-axis is the total number of configurations. For instance, to determine how many configurations perform within a 2× slowdown of Racket, read the y-axis where the corresponding line crosses 2 on the x-axis. Lines that climb steeply as they go to the right exhibit good performance on most configurations whereas lines that climb slowly signal poorer performance.

The first observation, based on the right-hand side of Figure 8, that we take away is that coercions reduce overheads in the benchmarks that cause long chains of proxies (quicksort, sieve, and n-body). This can be seen in the way the green line for coercions is to the left, sometimes far to the left, of the purple line for type-based casts. This demonstrates that catastrophic overheads are incidental (just a property of type-based casts and related technologies), and not inherent to gradual typing per se.

The second observation, based on the left-hand side of Figure 8, is that Typed Racket, with its contract-based runtime checks, and even with collapsible contracts[20], incurs significant incidental overheads. The yellow line for Typed Racket is far to the right of Grift on sieve, ray, and n-body. We hope that these benchmarks and results will help the developers of Typed Racket identify and eliminate these incidental overheads.

Third, it is interesting to compare the left-hand column to the right-hand column. Many researchers have speculated regarding whether fine-grained or coarse-grained gradual typing elicit more runtime overhead. The data suggests that there is not a simple answer to this question. From a syntactic point of view, it is certainly true that coarse-grained yields fewer opportunities for casts to be inserted. However, sometimes a single cast can have a huge impact on runtime, especially if it appears in a hot code region or if it wraps a proxy around a value that is used later in a hot code region. For example, in the sieve, ray, and n-body benchmarks, we see considerable overheads for Typed Racket on many coarse-grained configurations. On the other hand, fine-grained gradual typing provides many more opportunities for configurations to elicit more runtime overhead. For example, compare the left and right-hand sides for quicksort and sieve, in which there are catastrophic slowdowns for type-based casts in the fine-grained configurations but not in the coarse-grained configurations.

> **Answer to research question (2 c):** the overhead of Grift on partially typed code is no longer catastrophic, but there is still room for improvement.

4.4 Threats to Validity

One concern with these experiments is that the GTLC+ is a small language compared to other programming languages. For example, both Typed Racket and OCaml support separate compilation, tail-call optimization, unions, and polymorphism.The Grift compiler supports none of these. This is likely one of the reasons that Static Grift has performance on par with OCaml. Extending Grift to support these features will likely introduce overheads. The question relevant to this paper is whether there will be any additional overheads that arise from the interactions between gradual typing and the new features. For example, adding polymorphism with relational parametricity would require runtime sealing, which could incur significant overhead.

Another concern with being a small language is that the language features available in GTLC+ limit benchmarks that we are able to support. This has led to a numerically leaning suite of benchmarks. Of the 8 benchmarks presented in this paper, 6 feature a significant amount of arithmetic. There is a possibility that Grift performs really well on arithmetic benchmarks, but not as well on other types of benchmarks.

5 Future Work

Grift is still in it's infancy and more features are planned, such as modules, nominal records, type unions, and polymorphism. These features will bring the GTLC+ much closer to being a realistic programming language and will enable the evaluation of more benchmarks. Furthermore, there is plenty of room to improve performance and we identify three areas where we think improvements can be attained.

First, operations on references and arrays always check whether the address is proxied even in typed code regions,

which causes slowdowns in array intensive benchmarks. Monotonic reference [56] is an alternative approach that does not introduce such overheads. An implementation of this idea is currently being added to Grift and preliminary results show that it eliminates the overheads in statically-typed, array-intensive benchmarks.

Second, we believe that changing the in-memory representation of coercions could improve performance. The representation currently follows the grammar for coercions (Section 2), but if we instead followed the grammar for coercions in normal form, it would reduce the amount of space needed (by a constant factor) and there would be fewer memory indirections when composing coercions.

Finally, as mentioned above, Grift currently does not implement general-purpose optimizations common in mainstream compilers. It is yet to be shown if optimizations such as tail-call optimization, type inference, inlining, constant propagation, constant folding, and common subexpression elimination can improve the performance across the partially and/or dynamically typed configurations. We conjecture that such optimizations will eliminate many first-order checks, the main cause of slowdowns in dynamically typed code.

6 Conclusion

We have presented Grift, a compiler for exploring implementations of gradually typed languages. In particular, we implement and evaluate an important idea for efficient gradual typing: Henglein's coercions. Our experiments show that the performance of Grift on statically typed code is comparable to that of OCaml. For untyped code, Grift is slower than but in the same ballpark as Scheme implementations.

On partially typed code, our experiments show that space-efficient coercions eliminate the catastrophic slowdowns (i.e., changes to the time complexity) caused by sequences of casts without adding significant average-case overhead. We see significant speedups (10×) as 60% or more of a program is annotated with types. Our experiments show that the overhead in partially typed configurations with Grift is much lower than that with Typed Racket. According to the main author of Typed Racket, Typed Racket's design decisions are made in the context of a preexisting dynamically typed programming language, and therefore the current implementation introduces overhead to gradual typing that can be avoided in other systems [60].

The experiments reported here suggest a few conclusions. First, an implementation of a gradually typed programming language can deliver good performance on statically typed code. Second, catastrophic slowdowns are not an essential cost of gradual typing. Finally, some of the constant factors that are attributed to gradual typing are incidental to the current implementations of Gradual Typing. Furthermore, we believe that there is still incidental overhead in Grift that can be eliminated.

Acknowledgments

We would like to thank our shepherd Michael Greenberg, Sam Tobin-Hochstadt, Andrew Kent, and anonymous reviewers for their comments and suggestions. We also thank Frédéric Bour for his suggestions on improving our OCaml benchmarks. This material is based upon work supported by the National Science Foundation under Grant No. 1763922.

References

[1] 2017. https://flow.org/en/

[2] 2018. Gradual Typing Performance Benchmarks. https://pkgs.racket-lang.org/package/gtp-benchmarks

[3] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for All. In *Symposium on Principles of Programming Languages*.

[4] Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free: Parametricity, With and Without Types. In *International Conference on Functional Programming (ICFP)*.

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

[6] Andrew W. Appel. 1992. *Compiling with continuations*. Cambridge University Press, New York, NY, USA.

[7] Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. 2017. Sound Gradual Typing: Only Mostly Dead. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 54 (Oct. 2017), 24 pages. https://doi.org/10.1145/3133878

[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. *The PARSEC Benchmark Suite: Characterization and Architectural Implications*. Technical Report TR-811-08. Princeton University.

[9] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Lecture Notes in Computer Science, Vol. 8586. Springer Berlin Heidelberg, 257–281.

[10] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. 2009. Thorn: Robust, Concurrent, Extensible Scripting on the JVM. In *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. 117–136.

[11] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Softw. Pract. Exper.* 18, 9 (Sept. 1988), 807–820. https://doi.org/10.1002/spe.4380180902

[12] Luca Cardelli. 1983. *The Functional Abstract Machine*. Technical Report TR-107. AT&T Bell Laboratories.

[13] Luca Cardelli. 1984. Compiling a Functional Language. In *ACM Symposium on LISP and Functional Programming (LFP '84)*. ACM, 208–217.

[14] Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. In *International Conference on Functional Programming*.

[15] Olaf Chitil. 2012. Practical Typed Lazy Contracts. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 67–76.

[16] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. 1990. Combining Generational and Conservative Garbage Collection: Framework and Implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90)*. ACM, New York, NY, USA, 261–269. https://doi.org/10.1145/96709.96735

[17] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. 2011. Correct blame for contracts: no more scapegoating. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '11)*. ACM,

New York, NY, USA, 215–226.
[18] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *ESOP*.
[19] Marc Feeley. 2014. *Gambit-C: A portable implementation of Scheme*. Technical Report v4.7.2. Universite de Montreal.
[20] Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. Collapsible Contracts: Fixing a Pathology of Gradual Typing. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 133 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276503
[21] R. B. Findler and M. Felleisen. 2002. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*. 48–59.
[22] Robert Bruce Findler and Matthias Felleisen. 2002. *Contracts for Higher-Order Functions*. Technical Report NU-CCS-02-05. Northeastern University.
[23] Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, 303–315.
[24] P. Graham. 1995. *ANSI Common Lisp*. Prentice Hall.
[25] Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. 2005. Fine-grained interoperability through mirrors and contracts. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*. ACM Press, New York, NY, USA, 231–245.
[26] Michael Greenberg. 2014. Space-Efficient Manifest Contracts. *CoRR* abs/1410.2813 (2014). http://arxiv.org/abs/1410.2813
[27] Michael Greenberg. 2015. Space-Efficient Manifest Contracts. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 181–194. https://doi.org/10.1145/2676726.2676967
[28] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2010. Contracts Made Manifest. In *Principles of Programming Languages (POPL) 2010*.
[29] Ben Greenman and Zeina Migeed. 2018. On the Cost of Type-tag Soundness. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '18)*. ACM, New York, NY, USA, 30–39. https://doi.org/10.1145/3162066
[30] 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.
[31] Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. 2007. Relationally-Parametric Polymorphic Contracts. In *Dynamic Languages Symposium*.
[32] Lars T. Hansen and William D. Clinger. 2002. An Experimental Study of Renewal-older-first Garbage Collection. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM, New York, NY, USA, 247–258. https://doi.org/10.1145/581478.581502
[33] Anders Hejlsberg. 2012. Introducing TypeScript. Microsoft Channel 9 Blog.
[34] Fritz Henglein. 1994. Dynamic typing: syntax and proof theory. *Science of Computer Programming* 22, 3 (June 1994), 197–230.
[35] David Herman, Aaron Tomb, and Cormac Flanagan. 2007. Space-Efficient Gradual Typing. In *Trends in Functional Prog. (TFP)*. XXVIII.
[36] David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* 23, 2 (2010), 167–189.
[37] Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On Polymorphic Gradual Typing. In *International Conference on Functional Programming (ICFP)*. ACM.
[38] Lintaro Ina and Atsushi Igarashi. 2011. Gradual typing for generics. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA '11)*.
[39] Andrew W. Keep, Alex Hearn, and R. Kent Dybvig. 2012. Optimizing Closures in O(0)-time. In *Proceedings of the 2012 Workshop on Scheme and Functional Programming (Scheme '12)*.
[40] Jacob Matthews and Amal Ahmed. 2008. Parametric Polymorphism Through Run-Time Sealing, or, Theorems for Low, Low Prices!. In *Proceedings of the 17th European Symposium on Programming (ESOP'08)*.
[41] Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-Language Programs. In *The 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
[42] 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
[43] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 167–180. https://doi.org/10.1145/2676726.2676971
[44] Gregor Richards, Ellen Arteca, and Alexi Turcotte. 2017. The VM Already Knew That: Leveraging Compile-time Knowledge to Optimize Gradual Typing. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 55 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133879
[45] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete Types for TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*.
[46] Taro Sekiyama, Soichiro Ueda, and Atsushi Igarashi. 2015. Shifting the Blame - A Blame Calculus with Delimited Control. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*. 189–207. https://doi.org/10.1007/978-3-319-26529-2_11
[47] Jeremy G. Siek. 2008. Space-Efficient Blame Tracking for Gradual Types. (April 2008).
[48] Jeremy G. Siek and Ronald Garcia. 2012. Interpretations of the Gradually-Typed Lambda Calculus. In *Scheme and Functional Programming Workshop*.
[49] 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.
[50] Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*. 81–92.
[51] Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *European Conference on Object-Oriented Programming (LCNS)*, Vol. 4609. 2–27.
[52] Jeremy G. Siek, Peter Thiemann, and Philip Wadler. 2015. Blame and coercion: Together again for the first time. In *Conference on Programming Language Design and Implementation (PLDI)*.
[53] Jeremy G. Siek and Sam Tobin-Hochstadt. 2016. The Recursive Union of Some Gradual Types. In *Wadler Fest (LNCS)*, Sam Lindley, Conor McBride, Don Sannella, and Phil Trinder (Eds.). Springer.
[54] Jeremy G. Siek and Manish Vachharajani. 2008. Gradual Typing and Unification-based Inference. In *DLS*.
[55] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *SNAPL: Summit on Advances in Programming Languages (LIPIcs: Leibniz International Proceedings in Informatics)*.
[56] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015. Monotonic References for Efficient Gradual Typing. In *European Symposium on Programming (ESOP)*.
[57] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. Chaperones and impersonators: run-time support for reasonable interposition. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*.
[58] 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)*.

[59] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *Principles of Programming Languages (POPL)*. ACM.

[60] Sam Tobin-Hochstadt. 2019. Personal communication.

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

[62] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Symposium on Principles of Programming Languages*.

[63] Julien Verlaguet and Alok Menghrajani. [n. d.]. Hack: a new programming langauge for HHVM. https://code.facebook.com/posts/264544830379293/hack-a-new-programming-language-for-hhvm/

[64] Michael Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big Types in Little Runtime. In *Symposium on Principles of Programming Languages (POPL)*.

[65] Michael M. Vitousek, Jeremy G. Siek, Andrew Kent, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *Dynamic Languages Symposium*.

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

A Values, Macros, and Compose

Figures 10, 11, 12, and 13 lists the C structs use to represent values and macros used in the code examples to manipulate them. Figure 15 shows the code for the compose runtime function which follows the equations for coercion composition given in the next section, in Figure 17. It is worth mentioning that Figure 15 shows how the compiler handles recursive coercions while Figures 17 and 18 do not include their semantics. We plan to formalize the semantics in future work. Figure 14 gives the interface for an associative map/stack used in compose to recognize when we have already composed a recursive coercion that could be used for a particular pair of coercions.

```
#define TAG(value) (((int64_t)value)&0b111)
#define UNTAG_INT(value) (((int64_t)value)&~0b111)
#define TAG_INT(value, tag) (((int64_t)value)|tag)
#define UNTAG_REF(ref) ((obj*)UNTAG_INT(ref))
```

Figure 10. All allocated values have 3 bits that can be used for tagging.

```
typedef char* blame;
#define ID NULL
typedef struct {type type; blame info;} project_crcn;
typedef struct {type type} inject_crcn;
typedef struct {crcn fst; crcn snd} seq_crcn;
typedef struct {blame info} fail_crcn;
#define UNTAG_2ND(c) \
   ((struct {snd_tag second_tag;}*)UNTAG_INT(c))
typedef struct {
  snd_tag second_tag;
  int32_t arity; crcn ret;
  crcn args[] } fun_crcn;
typedef struct {
  snd_tag second_tag;
  crcn write; crcn read} ref_crcn;
typedef struct {
  snd_tag second_tag;
  int64_t size; crcn elems[] } tup_crcn;
typedef int64_t snd_tag;
typedef struct {
  snd_tag second_tag;
  crcn body[] } rec_crcn;
typedef union {
  project_crcn* prj;
  inject_crcn* inj;
  seq_crcn* seq;
  fail_crcn* fail;
  fun_crcn* fun;
  ref_crcn* ref;
  tup_crcn* tup;
  rec_crcn* rec} crcn;
// UNTAG_PRJ, UNTAG_FAIL, UNTAG_SEQ are similar to UNTAG_INJ
#define UNTAG_INJ(inj) ((inject_crcn)UNTAG_INT(inj))
// MK_SEQ, MK_PROJECTION, MK_INJECTION are similar
#define MK_REF_COERCION(r, w) \
   (tmp_rc = (ref_crcn*)GC_MALLOC(RC_SIZE),\
    tmp_rc->second_tag=REF_COERCION_TAG, \
    tmp_rc->read=r, tmp_rc->write=w,\
    (crcn)(TAG_INT(tmp_rc, HAS_2ND_TAG)))
```

Figure 12. Coercions are represented as directed graphs. The only back edges are recursive coercion nodes (`rec_coercion`). We maintain a normal form akin to the normal form shown in Figure 17 that isn't enforced by these types. Furthermore, we maintain the invariant that `rec_crcn` are only allocated if referenced by a subcoercion.

```
typedef struct {
  void* code;
  (obj)(*caster)(obj, type, type, blame);
  obj fvs[]; } closure;
typedef struct{obj elems[]} tuple;
#define MK_TUPLE(n) ((tuple*)GC_MALLOC(sizeof(obj) * n))
typedef struct{obj elem} box;
typedef struct{int64_t length; obj elems[]} vector;
#ifdef TYPE_BASED_CASTS
typedef struct {
  obj* ref;
  type source;
  type target;
  blame info;} ref_proxy;
#define MK_REF_PROXY(v, s, t, l) \
   (tmp_rp = (ref_proxy*)GC_MALLOC(RP_SIZE),\
    tmp_rp->value=v, tmp_rp->source=s,\
    tmp_rp->target=t, tmp_rp->info=l,\
    (obj)TAG_INT(tmp_rp, REF_PROXY_TAG)
#define UNTAG_FUN(fun) ((closure*)(fun)))
#elseif COERCIONS
#define UNTAG_FUN(fun) ((closure*)UNTAG_INT(fun))
typedef struct {obj* ref; crcn cast;} ref_proxy
#define MK_REF_PROXY(v, c) \
   (tmp_rp = (ref_proxy*)GC_MALLOC(RP_SIZE),\
    tmp_rp->value=v, tmp_rp->coerce=c,\
    (obj)TAG_INT(tmp_rp, REF_PROXY_TAG)
#endif
typedef struct {obj value; type source} nonatomic_dyn;
#define UNTAG_NONATOMIC(value) \
   ((nonatomic_dyn)UNTAG_INT(value))
typedef union {
  int64_t atomic;
  nonatomic_dyn* boxed} dynamic;
#define UNTAG(v) \
   ((TAG(v) == INT_TAG) ? (obj)(UNTAG_INT(v)>>3) : \
    (TAG(v) == UNIT_TAG) ? (obj)UNIT_CONSTANT : \
    ... (obj)UNTAG_NONATOMIC(v).value)
#define TYPE(v) \
   ((TAG(v) == INT_TAG) ? (type)INT_TYPE : \
    (TAG(v) == UNIT_TAG) ? (type)UNIT_TYPE :\
    ... UNTAG_NONATOMIC(v)->source)
#define INJECT(v, s) \
   ((s==INT_TYPE) ? TAG_INT(v<<3, INT_TAG) :\
    (s==UNIT_TYPE) ? DYN_UNIT_CONSTANT : ... \
    ... (tmp_na = (nonatomic_dyn*)GC_MALLOC(NA_DYN_SIZE),\
    tmp_na->value=value, tmp_na->source=s, (obj)tmp_na)
typedef union {
  int64_t fixnum; double flonum; dynamic dyn;
  closure* clos; tuple* tuple;
  box* box; vector* vec} obj;
```

Figure 13. Value Representation

```
typedef struct {
  crcn fst;
  crcn snd;
  crcn mapsto } assoc_triple;
typedef struct {
  unsigned int size;
  unsigned int next;
  assoc_triple *triples;} assoc_stack;

// Push a new triple on the assoc_stack
void
assoc_stack_push(assoc_stack *m, crcn f, crcn s, crcn t);

// return index of association of f and s
// returns -1 if the association isn't found
int grift_assoc_stack_find(assoc_stack *m, obj f, obj s);

// pop the most recent triple, return the mapsto value
obj
grift_assoc_stack_pop(assoc_stack *m);

// return the mapsto of the ith association
obj
grift_assoc_stack_ref(assoc_stack *m, int i);

// update the mapsto of the ith association
void
grift_assoc_stack_set(assoc_stack *m, int i, obj v);
```

Figure 14. The association stack is used `compose` to compose recursive equations at runtime. It associates two pointers as a key to another pointer.

```
#define TYPE_DYN_RT_VALUE 7
#define TYPE_INT_RT_VALUE 15
#define TYPE_BOOL_RT_VALUE 23
#define TYPE_UNIT_RT_VALUE 31
#define TYPE_FLOAT_RT_VALUE 39
#define TYPE_CHAR_RT_VALUE 47
typedef struct {int64_t index; int64_t hash } type_summary;
typedef struct {type_summary summary; type* body} ref_type;
typedef struct {type_summary summary;
  int64_t arity; type ret; type args[]} fun_type;
typedef struct {type_summary summary;
  int64_t size; type elems[]} tup_type;
typedef struct {type_summary summary;
  type* body} rec_type;
typedef union {
  int64_t atm;
  ref_type* ref; fun_type* fun;
  tup_type* tup; rec_type* rec} type;
```

Figure 11. Runtime types are either 64 bit integers or a pointer to a heap allocated type. Heap allocated types are hoisted and shared at runtime so that structural equality is equivalent to pointer equality. The lowest 3 bits of each type are used to distinguish between heap allocated types and atomic types. The `summary` field of heap allocated types is used in implementation hashconsing at runtime for the monotonic references implementation.

```
assoc_stack *as;
crcn compose(crcn fst, crcn snd, bool* id_eqv, int* fvs) {
  if (fst == ID) {if (snd == ID) return ID; else { *id_eqv = false; return snd; }}
  else if (snd == ID) { return fst; }
  else if (TAG(fst) == SEQUENCE_TAG) {
    sequence s1 = UNTAG_SEQ(fst);
    if (TAG(s1->fst) == PROJECT_TAG) {
      *id_eqv = false; return MK_SEQ(s1->fst, compose(s1->snd, snd, id_eqv, fvs)); }
    else if (TAG(snd) == FAIL_TAG) { *id_eqv = false; return snd; }
    else { sequence s2 = UNTAG_SEQ(snd);
      type src = UNTAG_INJ(s1->snd)->type; type tgt = UNTAG_PRJ(s2->fst)->type;
      blame lbl = UNTAG_PRJ(s2->fst)->lbl; crcn c = mk_crcn(src, tgt, lbl);
      bool unused = true;
      return compose(compose(seq->fst, c, &unused, fvs), s2->snd, id_eqv, fvs); }
  } else if (TAG(snd) == SEQUENCE_TAG) {
    if (TAG(fst) == FAIL) { *id_eqv = false; return fst; }
    else {
      crcn c = compose(fst, s2->fst, id_eqv, fvs);
      *id_eqv = false; return MK_SEQ(c, UNTAG_SEQ(seq)->snd); }
  } else if (TAG(snd) == FAIL) {
      *id_eqv = false; return (TAG(fst) == FAIL ? fst : snd); }
  } else if (TAG(fst) == HAS_2ND_TAG) {
    snd_tag tag1 = UNTAG_2ND(fst)->second_tag; snd_tag tag2 = UNTAG_2ND(fst)->second_tag;
    if (tag1 = REC_COERCION_TAG || tag2 = REC_COERCION_TAG) {
      int i = assoc_stack_find(as, c1, c2);
      if (i < 0) {
        assoc_stack_push(as, c1, c2, NULL); new_id_eqv = true;
        crcn c = (tag1 == REC_COERCION_TAG) ?
                    compose(REC_COERCION_BODY(c1), c2, &new_id_eqv, fvs):
                    compose(c1, REC_COERCION_BODY(c2), &new_id_eqv, fvs);
        crcn mu = assoc_stack_pop(as);
        if (!*new_id_eqv) *id_eqv = false;
        if (mu == NULL) return c;
        *fvs -= 1;
        if (*fvs = 0 && new_id_eqv) { return ID }
        else { REC_COERCION_BODY_INIT(mu, c); return mu; }
      else { crcn mu = assoc_stack_ref(as, i);
        if (mu = NULL) {*fvs += 1; mu = MK_REC_CRCN();
          assoc_stack_set(as, i, mu); return mu; }
        else { return mu; }}}
    else if (tag1 == FUN_COERCION_TAG) {
      return compose_fun(fst, snd); }
    else if (tag1 == REF_COERCION_TAG) {
      ref_crcn r1 = UNTAG_REF(fst);
      ref_crcn r2 = UNTAG_REF(snd);
      if (read == ID && write == ID) return ID;
      else {
        crcn c1 = compose(r1->read, r2->read);
        crcn c2 = compose(r2->write, r1->write);
        return MK_REF_COERCION(c1, c2); }}
    else { // Must be tuple coercions
      return compose_tuple(fst, snd); }}
  else { raise_blame(UNTAG_FAIL(fst)->lbl); }
}
```

Figure 15. The compose function for normalizing coercions.

B Semantics of a Gradual Language

Figure 17 gives the semantics GTLC+ for the GTLC+ including functions, proxied references, and pairs but excluding equirecursive types. The type system is the standard one for the gradually typed lambda calculus [36, 47, 50]. The operational semantics, as usual, is expressed by a translation to an intermediate language with explicit casts.

Consider the source program in Figure 16 which calculates the value 42 by applying the `add1` function, by way of variable `f`, to the integer value 41. The type of `add1` does not exactly match the type annotation on `f` (which is `Dyn` $\rightarrow$ `Dyn`) so the compiler inserts the cast:

```
(cast add1 (Int → Int) (Dyn → Dyn) 12)
```

The application of `f` to `42` requires a cast on `42` from `Int` to `Dyn`. Also, the return type of `f` is `Dyn`, so the compiler inserts a cast to convert the returned value to `Dyn` to satisfy the type ascription.

As mentioned in section 2 we consider to approaches to the implementation of runtime casts: traditional space-inefficient casts which we refer to as *type-based casts* and space-efficient *coercions*. For type-based casts, the dynamic semantics that we use is almost covered in the literature. We use the lazy-D cast semantics which is described by Siek and Garcia [48]. (They were originally described using coercions by Siek et al. [49].) The distinction between lazy-D and the more commonly used lazy-UD semantics [66] is not well-known, so to summarize the difference: in lazy-D, arbitrary types of values may be directly injected into type Dyn, whereas in lazy-UD, only values of a *ground* type may be directly injected into Dyn. For example, Int and Dyn $\rightarrow$ Dyn are ground types, but Int $\rightarrow$ Int is not.

The one missing piece for our purposes are the reduction rules for proxied references, which we adapt from the coercion-based version by Herman et al. [36]. In this setting, proxied references are values of the form $(v : \mathsf{Ref}\ T_1 \Rightarrow^{\ell} \mathsf{Ref}\ T_2)$. The following are the reduction rules for reading and writing to a proxied reference.

$$!(v : \mathsf{Ref}\ T_1 \Rightarrow^{\ell} \mathsf{Ref}\ T_2) \longrightarrow\ !v : T_1 \Rightarrow^{\ell} T_2$$

$$(v_1 : \mathsf{Ref}\ T_1 \Rightarrow^{\ell} \mathsf{Ref}\ T_2) := v_2 \longrightarrow v_1 := (v_2 : T_2 \Rightarrow^{\ell} T_1)$$

Regarding coercions, the dynamic semantics that we used is less well-covered in the literature. Again, we use the lazy-D semantics of Siek et al. [49], but that work, despite using coercions, did not define a space-efficient semantics. On the other hand, Siek et al. [52] give a space-efficient semantics with coercions, but for the lazy-UD semantics. To this end, they define a normal form for coercions and a composition operator that compresses coercions. Here we adapt that approach to lazy-D, which requires some changes to the normal forms and to the composition operator. Also, that work did not consider mutable references, so here we add support for references.

Figure 17 defines a representative subset of the types and coercions used in Grift's intermediate language. The figure also defines the *meet* operation and the *consistency* relation on types and the *composition* operator on coercions.

Space-efficient coercions are defined by a grammar consisting of three rules that enable coercion composition by the composition operator defined in Figure 17. Let c, d range over space-efficient coercions, i range over final coercions, and g range over middle coercions. Space-efficient coercions are either the identity coercion ι, a projection followed by a final coercion $(I?^p\ ;\ i)$, or just a final coercion. A final coercion is either the failure coercion $\bot^{IpJ}$, a middle coercion followed by an injection $(g\ ;\ I!)$, or just an intermediate coercion. An intermediate coercion is either the identity coercion ι, the function coercion $c \rightarrow d$, the tuple coercion $c \times d$, the reference coercion Ref c d, where c is applied when writing and d is applied when reading. The main difference between the lazy-D coercions shown here and those of Siek et al. [52] is in the injection $I!$ and projection $J?^p$ coercions, which take any *injectable* type (anything but Dyn) instead of only ground types. This change impacts the coercion composition operation, in the case where an injection $I!$ meets a projection $J?^p$ we make a new coercions whose source is I and target is J with the coercion creation operation $\langle I \Rightarrow^p J \rangle$ (Figure 17).

The following is the syntax of Grift's intermediate language.

$$\begin{aligned}
u &::= k \mid a \mid \lambda x.M \mid (u, u) \\
v &::= u \mid (v, v) \mid u\langle g\ ;\ I! \rangle \mid u\langle c \rightarrow d \rangle \\
b &::= \mathtt{blame}\, p \mid \mathtt{error} \\
M, N &::= b \mid v \mid x \mid M\,N \mid (M, M) \mid (\mathtt{fst}\, M) \mid (\mathtt{snd}\, M) \mid \\
&\quad M\langle c \rangle \mid \mathtt{ref} M \mid !M \mid M{:=}N
\end{aligned}$$

Figure 18 defines the dynamic semantics. The language forms `ref`M, $!M$, and $M{:=}N$ are for allocating, dereferencing, and updating a proxied pointer, respectively. Regarding the definition of values, the value form $u\langle \mathsf{Ref}\ c\ d \rangle$ represents a proxied reference whereas an address a is a regular reference.

The dynamic semantics is given by four reduction relations: cast reductions, stateless reductions, statefull reductions, and configuration reduction. The heap μ maps an address to a value.

The cast reduction rules define the semantics of applying a cast to a value. Space efficiency is achieved with the reduction that takes a coerced value $u\langle i \rangle$ and a coercion c and compresses the two coercions to produce the coerced value $u\langle i \fatsemi c \rangle$.

Regarding the statefull reduction rules, for dereferencing a reference there are two rules, one for raw addresses and the other for a proxy. Thus, an implementation must dispatch on the kind of reference. If it's an address, the value is loaded from the heap. If it's a proxy, the underlying reference is dereferenced and then the proxy's read coercion c is applied. The story is similar for writing to a proxied reference.

Source Program:

```
(let ([add1 : (Int → Int)
       (lambda ([x : Int]) (+ x 1))])
  (let ([f : (Dyn → Dyn) add1])
    (ann (f 41) Int)))
```

After Cast Insertion:

```
(let ([add1 (lambda (x) (+ x 1))])
  (let ([f (cast add1 (Int → Int) (Dyn → Dyn) L1)])
    (cast (f (cast 41 Int Dyn L2)) Dyn Int L3)))
```

Figure 16. An example of the Grift compiler inserting casts. The `L1`, `L2`, etc. are blame labels that identify source code location.

Types and coercions

Base Types	B	::=	$\mathsf{Int} \mid \mathsf{Bool} \mid \ldots$
Injectables	I, J	::=	$B \mid T \to T \mid T \times T \mid \mathsf{Ref}\ T$
Types	T	::=	$\mathsf{Dyn} \mid I$
Coercions (in normal form)	c, d	::=	$\iota \mid (I?^p\ ;\ i) \mid i$
Final Coercions	i	::=	$\bot^{IpJ} \mid (g\ ;\ I!) \mid g$
Middle Coercions	g	::=	$\iota \mid c \to d \mid c \times d \mid \mathsf{Ref}\ c\ d$
Identity-free Coercions	f	::=	$(I?^p\ ;\ i) \mid (g\ ;\ I!) \mid c \to d \mid c \times d \mid \mathsf{Ref}\ c\ d \mid \bot^{IpJ}$

Consistency $\boxed{T \sim T}$

$$\mathsf{Dyn} \sim T \qquad T \sim \mathsf{Dyn} \qquad B \sim B$$

$$\frac{T_1 \sim T_2}{\mathsf{Ref}\ T_1 \sim \mathsf{Ref}\ T_2}$$

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

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

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

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

$$\begin{aligned}
(B \Rightarrow^l B) = (\mathsf{Dyn} \Rightarrow^l \mathsf{Dyn}) &= \iota \\
(I \Rightarrow^l \mathsf{Dyn}) &= \iota\ ;\ I! \\
(\mathsf{Dyn} \Rightarrow^l I) &= I?^l\ ;\ \iota \\
(T_1 \to T_2 \Rightarrow^l T_1' \to T_2') &= (T_1' \Rightarrow^l T_1) \to (T_2 \Rightarrow^l T_2') \\
(T_1 \times T_2 \Rightarrow^l T_1' \times T_2') &= (T_1 \Rightarrow^l T_1') \times (T_2 \Rightarrow^l T_2') \\
(\mathsf{Ref}\ T \Rightarrow^l \mathsf{Ref}\ T') &= \mathsf{Ref}\ (T' \Rightarrow^l T)\ (T \Rightarrow^l T')
\end{aligned}$$

Coercion composition $\boxed{c \fatsemi d = r}$

$$\begin{aligned}
(g\ ;\ I!) \fatsemi (J?^p\ ;\ i) &= g \fatsemi (\langle I \Rightarrow^p J \rangle \fatsemi i) \\
c \to d \fatsemi c' \to d' &= (c' \fatsemi c) \to (d \fatsemi d') \\
c \times d \fatsemi c' \times d' &= (c \fatsemi c') \times (d \fatsemi d') \\
\mathsf{Ref}\ c\ d \fatsemi \mathsf{Ref}\ c'\ d' &= \mathsf{Ref}\ (c' \fatsemi c)\ (d \fatsemi d')
\end{aligned}$$

$$\begin{aligned}
(I?^p\ ;\ i) \fatsemi c &= I?^p\ ;\ (i \fatsemi c) \\
g_1 \fatsemi (g_2\ ;\ I!) &= (g_1 \fatsemi g_2)\ ;\ I! \\
\iota \fatsemi c = c \fatsemi \iota &= c \\
g \fatsemi \bot^{IpJ} = \bot^{IpJ} \fatsemi c &= \bot^{IpJ}
\end{aligned}$$

Figure 17. Types, coercions, and their operations.

Runtime Structures

$$
\begin{array}{lcl}
\mu & ::= & \emptyset \mid \mu(a \mapsto v) \\
\mathcal{E} & ::= & \mathcal{F} \mid \mathcal{F}[\Box\langle f \rangle] \\
\mathcal{F} & ::= & \Box \mid \mathcal{E}[\Box\, M] \mid \mathcal{E}[v\, \Box] \mid \mathcal{E}[(\Box, M)] \mid \mathcal{E}[(v, \Box)] \mid \\
 & & \mathcal{E}[(\mathtt{fst}\, \Box)] \mid \mathcal{E}[(\mathtt{snd}\, \Box)] \mid \mathcal{E}[\mathtt{ref} \Box] \mid \mathcal{E}[!\Box] \mid \\
 & & \mathcal{E}[\Box := M] \mid \mathcal{E}[v := \Box]
\end{array}
$$

Cast reduction rules $\boxed{M \longrightarrow_c N}$

$$
\begin{array}{rcl}
\mathcal{F}[u\langle \iota \rangle] & \longrightarrow_c & \mathcal{F}[u] \\
\mathcal{F}[u\langle i \rangle\langle c \rangle] & \longrightarrow_c & \mathcal{F}[u\langle i \fatsemi c \rangle] \\
\mathcal{F}[(u, u')\langle c \times d \rangle] & \longrightarrow_c & \mathcal{E}[(u\langle c \rangle, u'\langle d \rangle)] \\
\mathcal{F}[u\langle \bot^{IpJ} \rangle] & \longrightarrow_c & \mathtt{blame}\, p
\end{array}
$$

Stateless reduction rules $\boxed{M \longrightarrow_e N}$

$$
\begin{array}{rcll}
\mathcal{E}[(\lambda x.M)\, v] & \longrightarrow_e & \mathcal{E}[[x := v]M] & \\
\mathcal{E}[u\langle c \to d \rangle\, v] & \longrightarrow_e & \mathcal{E}[u\, (v\langle c \rangle)\langle d \rangle] & \\
\mathcal{E}[(\mathtt{fst}\, (v, v'))] & \longrightarrow_e & \mathcal{F}[v] & \\
\mathcal{E}[(\mathtt{snd}\, (v, v'))] & \longrightarrow_e & \mathcal{F}[v'] & \\
\mathcal{E}[\mathtt{blame}\, p] & \longrightarrow_e & \mathtt{blame}\, p & \text{if } \mathcal{E} \neq \Box \\
\mathcal{E}[\mathtt{error}] & \longrightarrow_e & \mathtt{error} & \text{if } \mathcal{E} \neq \Box
\end{array}
$$

Statefull reduction rules $\boxed{M, \mu \longrightarrow_s N, \mu}$

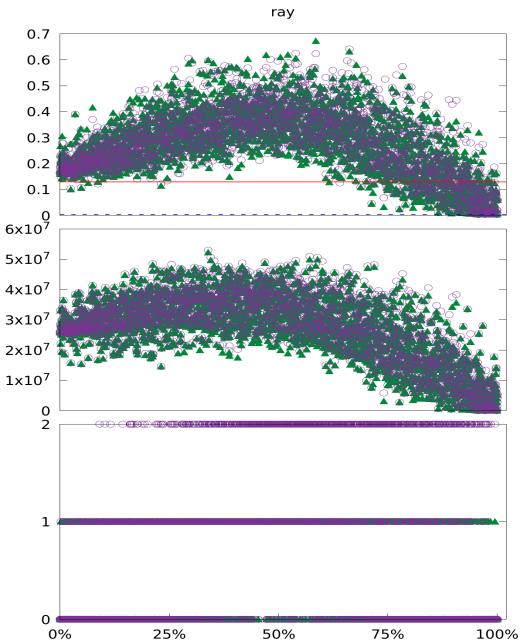
$$
\begin{array}{rcll}
\mathcal{E}[\mathtt{ref} v], \mu & \longrightarrow_s & \mathcal{F}[a], \mu(a \mapsto v) & \text{if } a \notin \mathrm{dom}(\mu) \\
\mathcal{E}[!a], \mu & \longrightarrow_s & \mathcal{F}[\mu(a)], \mu & \\
\mathcal{E}[!(a\langle \mathrm{Ref}\ c\ d \rangle)], \mu & \longrightarrow_s & \mathcal{F}[(!a)\langle d \rangle], \mu & \\
\mathcal{E}[a := v], \mu & \longrightarrow_s & \mathcal{F}[a], \mu(a \mapsto v) & \\
\mathcal{E}[a\langle \mathrm{Ref}\ c\ d \rangle := v], \mu & \longrightarrow_s & \mathcal{E}[a := v\langle c \rangle], \mu &
\end{array}
$$

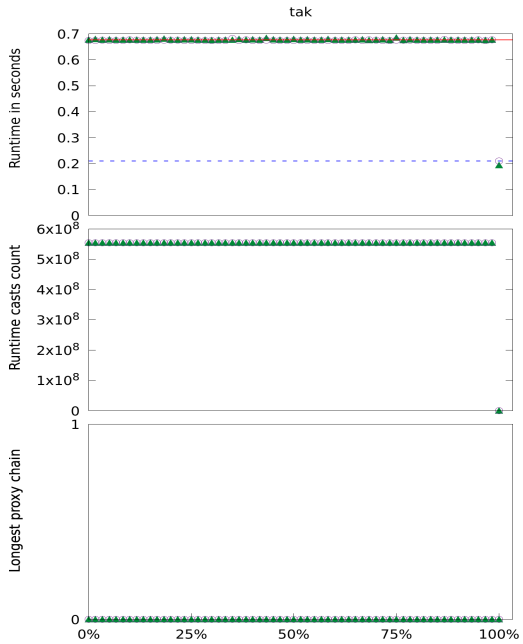
Configuration reduction rules $\boxed{M, \mu \longrightarrow N, \mu}$

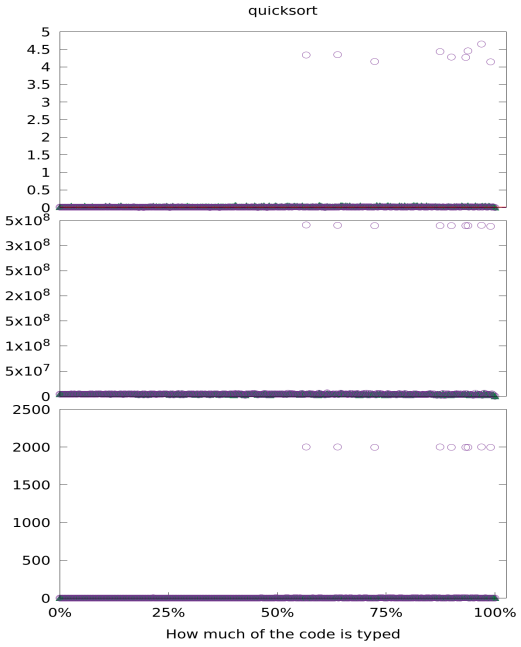
$$
\frac{M \longrightarrow_X N \qquad X \in \{c, e\}}{M, \mu \longrightarrow N, \mu} \qquad \frac{M, \mu \longrightarrow_s N, \mu'}{M, \mu \longrightarrow N, \mu'}
$$

Figure 18. Semantics of the intermediate language of Grift.

Regarding the configuration reduction rules, they simply enable the other three classes of rules and pass along the heap as appropriate.







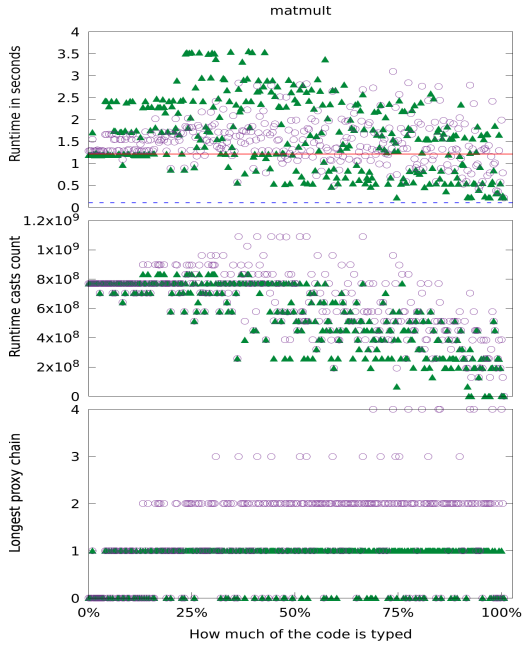


Figure 20

Figure 19

C More Performance Evaluation

Figures 19 and 20 give the results for the rest of the benchmarks that were not included in Figure 7.