An Efficient Compiler for the Gradually Typed Lambda Calculus

ANDRE KUHLENSCHMIDT, Indiana University
DEYAAELDEEN ALMAHALLAWI, Indiana University
JEREMY G. SIEK, Indiana University

Gradual typing combines static and dynamic typing in the same program. One would hope that the performance in a gradually typed language would range between that of a dynamically typed language and a statically typed language. Existing implementations of gradually typed languages have not achieved this goal due to overheads associated with runtime casts. Takikawa et al. (2016) report up to 100× slowdowns for partially typed programs. In this paper we present a compiler, named Grift, for evaluating implementation techniques for gradual typing. We take a straightforward but unexplored implementation approach for gradual typing, that is, ahead-of-time compilation to native assembly code using space-efficient coercions.

Our experiments show that this approach achieves performance on par with OCaml on statically typed programs but is slower than other dynamically typed languages on untyped programs. On partially typed code, the speedup with respect to untyped code ranges from 0.42× to 19.90× across the benchmarks. We implement casts using coercions and show that coercions eliminate catastrophic slowdowns without introducing significant overhead in all benchmarks. Across the benchmarks, coercions range from 9% slower (fft) to 13.65× faster (quicksort) than regular casts. We also implement the monotonic references and show that they eliminate all overhead in statically typed code. For partially typed code, they are faster on average than the traditional proxied references, sometimes up to 1.65×.

1 INTRODUCTION

Gradual typing combines static and dynamic type checking in the same program, giving the programmer control over which typing discipline is used for each region of code [Anderson and Drossopoulou 2003; Gronski et al. 2006; Matthews and Findler 2007; Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006]. We would like gradually typed languages to be *efficient*, *sound*, and provide seamless *interoperability*. Regarding efficiency, the performance of gradual typing should range from being similar to that of a dynamically typed language to that of a statically typed language. Regarding soundness, programmers and compilers would like to know that runtime values respect their compile-time types. Third, regarding interoperability, static and dynamic regions of code should interoperate seamlessly. More specifically, as a programmer adds or removes correct type information, the behavior of programs should not change, i.e., the *gradual guarantee* of Siek et al. [2015b].

While there is considerable literature on the theory of gradual typing, the research community is still in the early stages of exploring implementation technologies for gradual typing. When comparing these technologies, it is important to realize that the experiments to-date have been conducted in a variety of languages with quite different semantics and choices regarding efficiency, soundness, and interoperability. Some combination of choices have been explored, and others have not. Figure 1 presents a summary of the languages implemented in recent research on gradual typing and it includes the language that we study in this paper, an extension to the Gradually Typed Lambda Calculus (GTLC+). A full (black) circle means that the language satisfies the given property and an empty (white) circle means that it does not. A dash means that the property is not applicable, typically because the language does not support the feature in question. Our work is the first empirical study of the implementation technologies required to achieve space-efficiency in the context of a language with sound gradual typing and structural types (.e.g. functions, tuples, etc.). We explain the table in detail in the following paragraphs.

Authors' addresses: Andre Kuhlenschmidt, Indiana University, akuhlens@indiana.edu; Deyaaeldeen Almahallawi, Indiana University, dalmahal@indiana.edu; Jeremy G. Siek, Indiana University, jsiek@indiana.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

		Gradual Guarantee wrt.		
Language	Sound	structural types	nominal types	Space-Efficient
TypeScript	○	●	●	●
Safe TypeScript	●	○	○	●
Nom	●	–	●	●
Reticulated Python	◐	●	●	●
Typed Racket	●	●	●	◐
GTLC+	●	●	–	●

Fig. 1. A comparison of gradually typed programming languages.

The semantics of TypeScript [Bierman et al. 2014; Hejlsberg 2012] is given via type erasure, so it does not provide soundness and its performance is on par with dynamic languages but not static ones. However, TypeScript does provide seamless interoperability. Safe TypeScript[Rastogi et al. 2015] addresses the soundness concerns by restricting the static type system, which results in loss of the gradual guarantee. Richards et al. [2017] address performance concerns for Safe TypeScript by adapting the Higgs JIT compiler and virtual machine (VM) [Chevalier-Boisvert and Feeley 2015], re-purposing the VM's notion of *shape* to lower the overhead of implementing monotonic objects. Richards et al. [2017] reports that this techniques reduces the worst slowdowns to 45%, with an average slowdown of just 7%.

Meanwhile, Muehlboeck and Tate [2017] show that many of the efficiency concerns related to gradual typing do not apply to nominally-typed object-oriented languages without generics. They implement such a language, named Nom, and demonstrate that their implementation has low performance overhead.

Reticulated Python of Vitousek et al. [2017, 2014] supports a partial form of soundness without using proxies by a defense-in-depth approach called *transient*. Reticulated Python supports the gradual guarantee and is obviously space-efficient (no proxies) but it does incur a significant constant-factor time overhead in statically typed code.

The design of Typed Racket [Tobin-Hochstadt and Felleisen 2008] is the most closely related to the GTLC+ of this paper. Typed Racket supports sound gradual typing and provides the gradual guarantee for both structural and nominal types. However, the implementation of Typed Racket suffers from slowdowns of up to 100× [Takikawa et al. 2015, 2016] on a partially typed programs, partly because it does not implement space efficient casts. Bauman et al. [2017] demonstrate that Pycket, a tracing JIT, can eliminate 90% of the overheads in Typed Racket. However, Pycket does not address the space-leaks pointed out by Herman et al. [2007]. The concurrent work of Feltey et al. [2018] addresses the space-efficiency concerns for mutable references but not functions.

In this paper we present evidence that efficient gradual typing can be achieved in structurally-typed functional languages by relatively straightforward means. We build and evaluate an ahead-of-time compiler that uses carefully chosen runtime representations and implements two important ideas from the theory of gradual typing. It uses space efficient *coercions* [Garcia 2013; Herman et al. 2010; Siek et al. 2015a; Siek and Wadler 2009] to implement casts and it reduces overhead in statically typed code by using *monotonic references* [Siek et al. 2015c].

Road Map. The paper continues as follows.

- Section 2 provides background on gradual typing.
- Section 3 describes the implementation of the compiler.
- Section 4.2 shows that coercions eliminate catastrophic slowdowns without adding significant overhead.
- Section 4.3 shows that monotonic references eliminate overhead in statically typed code.
- Section 4.4 makes comparisons to other programming languages to provide context for the performance of Grift on typed and untyped benchmarks.

2 THE PERFORMANCE CHALLENGE OF SOUND, STRUCTURAL 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 [Anderson and Drossopoulou 2003; Gronski et al. 2006; Siek and Taha 2006]. The dynamic semantics for gradual typing is based on the semantics of *contracts* [Findler and Felleisen 2002a; Gray et al. 2005], *coercions* [Henglein 1994], and *interlanguage migration* [Matthews and Findler 2007; Tobin-Hochstadt and Felleisen 2006]. 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 [Chitil 2012; Dimoulas et al. 2011, 2012; Greenberg 2015; Greenberg et al. 2010; Guha et al. 2007; Matthews and Ahmed 2008; Strickland et al. 2012]. In the following we give a brief introduction to gradual typing, and intuition for the problems we address in this work.

Consider the classic example of Herman et al. [2007] shown in Figure 2. The pair of mutually recursive functions, `even?` and `odd?`, are written in a variant of Typed Racket that we have extended to support fine-grained gradual typing. This example uses continuation passing style to concisely illustrate efficiency challenges in gradual typing. While this example is contrived, we shall see the same problems occur in real programs under complex situations. ([Takikawa et al. 2016] also report programs that exhibit these problems.) On the left side of the figure we have a partially typed function, named `even?`, which checks if an integer is even. On the right side of the figure we have a fully typed function, named `odd?`, which 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 casted 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 performs any conversion needed between the representation of a dynamic values and integers. Conversely, the value `#t` is casted to `Dyn` because it is used as the argument to a function that expects `Dyn`. This 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*, and satisfy the consistency relationship. 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 [Findler and Felleisen 2002a], but Herman et al. [2007] 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 our example, Figure 4 shows the runtime, number of casts performed, and longest chain of proxies for this example under the title of `cps-even-odd`. From the graph concerning longest proxy chains, we can see that type-based casts accumulate longer chains of proxies as we increase parameter `n`. The coercions of Henglein [1994] solve this space problem by providing a representation that enables the compression of higher-order casts (casts that require proxies) as suggested by Herman et al. [2010]. Coercions use a constant amount of space by doing additional work at each step to compress the casts into a single proxy.

This unbounded space overhead 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 auxilliary `partition!` and `swap!` functions. Like function types, reference types require proxying to apply casts during read and write operations. Again, a naive implementation of casts allows the proxies to

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

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

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

```
(define sort!
  : ((Vectorof Int) Int Int -> ())
  (λ ([v : (Vectorof 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!
  : ((Vectorof Int) Int Int -> ())
  (λ ([v : (Vector 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!
  : ((Vectorof Int) Int Int -> Int)
  (λ ([v : (Vectorof 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))))
```

Fig. 3. The **sort!** function implements the Quicksort algorithm.

accumulate; each recursive call to **sort!** causes a cast that can add 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. $O(n^3)$.

3 THE GRIFT COMPILER

The Grift compiler takes programs in an extended version of the gradually typed lambda calculus (GTLC+) and compiles them to C, using the Clang compiler to generate x86 executables. The Clang compiler provides low level optimizations. The language includes base types such as fixnums, double precision floats, and bools but does not support implicit conversions between base types (i.e. no numeric tower). The GTLC+ also include structural types such as n-ary tuples, mutable vectors, and higher-order functions. It does not yet implement space-efficient tail recursion. The operational semantics for the GTLC+ and a more detailed description of the Grift compiler are

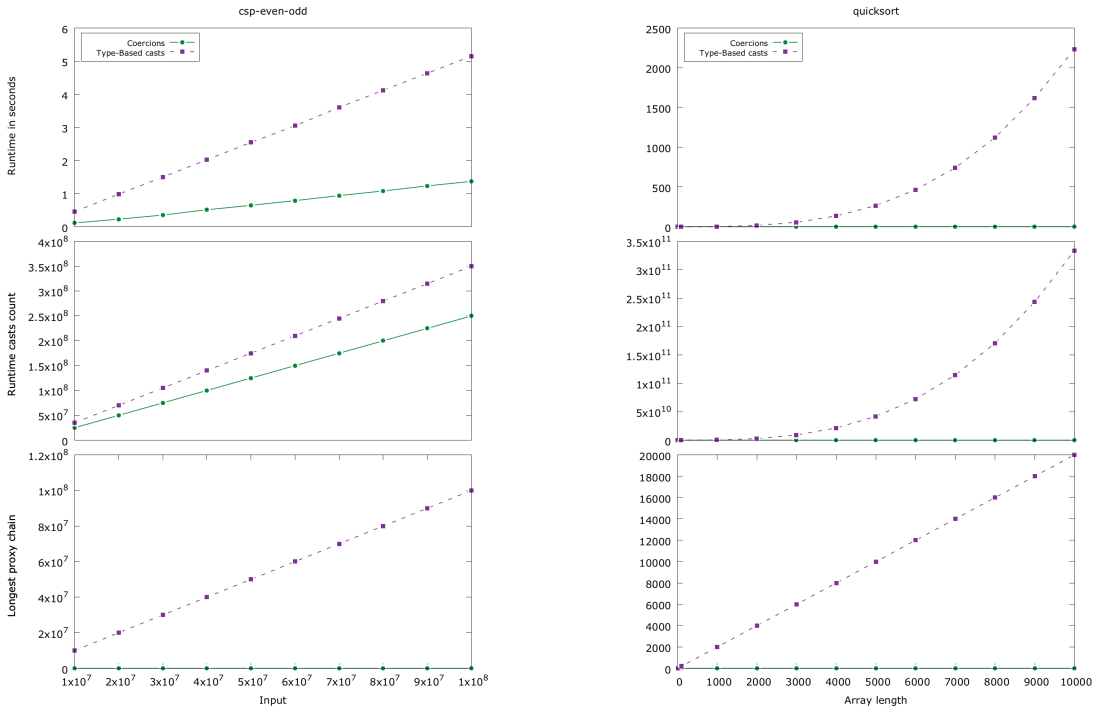


Fig. 4. The runtime, number of casts, and longest proxy chain that occur as we vary input size for the even/odd and quicksort examples in Figures 2 and 3. The graphs for longest proxy chain show that space-efficient coercions compress casts and thus operate in constant space. The runtime graph for quicksort shows that space-inefficiency can also increase the asymptotic time complexity of an algorithm.

available from Kuhlenschmidt et al. [2018]. This section presents a high level description of the techniques used to generate code for coercions. The code is available online at the following URL:

https://github.com/Gradual-Typing/Grift

The first step in the Grift compiler is to translate to an intermediate language with explicit casts. This process is standard [Herman et al. 2010; Siek 2008; Siek and Taha 2006] except for the optimizations that we implement to avoid unnecessary casts in untyped code [Kuhlenschmidt et al. 2018].

The next step in the compiler is exposing 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 and monotonic references in Section 3.3. After lowering casts, Grift performs closure conversion using a flat representation [Appel 1992; Cardelli 1983, 1984], and translates all constructors and destructors to memory allocations, reads, writes, and numeric manipulation.

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 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 proxied 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 monotonic reference type is a pointer to values stored in the reference and the runtime type information of the values. 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 atomic types (e.g. Int and Bool), the injected value is stored in the other 61 bits. For non-atomic types, the other 61 bits are a pointer to a pair of 64-bit items that contain the injected value and its type. In the following, the macros for manipulating 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 heap allocated values that indicate how to perform a cast at runtime. We refer the reader to the literature for an introduction to coercions [Henglein 1994; Siek and Garcia 2012; Siek et al. 2015a]. In Grift, the coercions that can be inferred from static type information at compile time are allocated once at the start of the program. At runtime, the generated code calls the C function `coerce`, described below, to apply a coercion to a value, that is, to interpret a coercion.

Coercions are represented as 64-bit values where the lowest 3 bits indicate whether the coercion is a projection, injection, sequence, failure, or identity. The number of pointer tags is limited, so the rest of the coercions are identified by a tag stored in the first word of their structures. For an identity coercion, the remaining 61 bits are not used. For the other coercions, the remaining 61 bits stores a pointer to a heap-allocated structure that depends on the kind of coercion:

- 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 of the projection 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 of the injected value.
- Function coercions ($c_1, \ldots, c_n \to c_r$) 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.
- Proxied reference ($\text{Ref}_p\ c_1\ c_2$) coercions cast between two proxied reference types and are represented as 3×64 bits, including the secondary tag, a coercion for writing, and another coercion for reading.
- Monotonic reference coercions ($\text{Ref}_m\ T$) cast between two monotonic reference types and are represented in 2×64 bits, including the secondary tag and a type.
- Sequences coercions ($c_1\ ;\ c_2$) store two coercions in 2×64 bits.
- Failure coercions ($\bot^p$) 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 5, 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: {
    sec_tag tag = UNTAG_2ND(c)->second_tag;
    if (tag == REF_COERCION_TAG) {
      if (TAG(v) != REF_PROXY) {
        return MK_REF_PROXY(v, c); }
      else { ref_proxy p = UNTAG_PREF(v);
        crcn c2 = compose(p->coercion, c);
        return MK_REF_PROXY(p->ref, c2); }
    } else if (tag == 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 FAIL_TAG: raise_blame(UNTAG_FAIL(c)->lbl);
  }
}
```

Fig. 5. The coerce function applies a coercion to a value.

When coercing a function, coerce checks whether the function has previously been coerced. If so, 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 (Appendix A Figure 11). The call to compose is what maintains space efficiency. If the function has not been previously coerced, then we access its function pointer for casting and apply that to the function and the coercion that needs to be applied. This "caster" function allocates and initializes a proxy closure. Coercing a proxied reference 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. Failure coercions halt execution and report an error.

Applying Functions. Because the coercions 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 [2012] which itself is inspired by Findler and Felleisen [2002b].

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. Since coercions are space efficient 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.

3.3 Implementation of Monotonic References

A monotonic heap cell has two parts; the first stores runtime type information (RTTI), and the second stores the value. Grift generates pointer dereference and write instructions for reading and writing a fully statically-typed monotonic reference. Otherwise, the value being read or written has to be cast. The details of the latter process and that of casting an address is described below.

To cast a monotonic reference, we cast the underlying value on the heap. First, the RTTI is read from the first word pointed to by the address. The address is returned if the RTTI equals the target type of the cast. However the equality check can be expensive if implemented naively because the structures of both types will be traversed. Instead, we hashcons [Allen 1978] all types to reduce structural equality to pointer equality. If the check fails, we call `tglb`, which computes the greatest lower bound of two types, and then overwrite the RTTI with the result. Next we cast the values in the reference from the old RTTI to the new RTTI. After the cast returns, we check if the current RTTI is the same as the one we wrote earlier to the heap and write the new value to the heap only if they are indeed equal. Otherwise, a value with a more precise type has already been written to the heap so we discard the current value and return the address.

Reading from a non-static reference proceeds as follows: the value is read from the second word pointed to by the address, the RTTI is read from the first word, then we cast the read value from the RTTI type to the static type. For writing, the RTTI will be read first from the heap and then we cast between the static type and the RTTI. Again, we check if the RTTI has changed during the casting process, if yes, we drop the casted value, otherwise, we write the new value to the heap.

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 cost of achieving space efficiency with coercions?** (Section 4.2)
(2) **How do monotonic references compare with proxied references?** (Section 4.3)
(3) **What is the overhead of gradual typing?** (Sec. 4.4) We subdivide this question into the overheads on programs that are (a) statically typed, (b) untyped, and (c) partially typed.

4.1 Experimental Methodology

In these experiments we use benchmarks from a number of sources: the well-known Scheme benchmark suite (R6RS) used to evaluate the Larceny [Hansen and Clinger 2002] and Gambit [Feeley 2014] compilers, the PARSEC benchmarks [Bienia et al. 2008], and the Computer Language Benchmarks Game (CLBG). 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 performance similar to the benchmarks of Takikawa et al. [2015, 2016]. The benchmarks that we use are:

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)`. A test of non-tail calls and arithmetic.

ray (R6RS) Ray tracing a scene, 20 iterations. A test of floating point arithmetic adapted from Graham [1995].

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 in the comparisons to other languages (Sec. 4.4.1 and 4.4.2) and 200×200 for the evaluation of partially typed programs (Sec. 4.2 and 4.3).

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 $65,536$ real-valued points. A test of floating point.

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

Porting Benchmarks. In all the benchmarks we have to port the benchmarks to the GTLC. For the R6RS and CLBG benchmarks, ports add types and convert tail recursive loops to an iterative form. For the Blackscholes benchmark, we use 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 our external comparisons alter the original benchmarks as described in Section 4.4 to account for these differences. The source code for all benchmarks are available online [1].

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 5.0.0, the Gambit compiler is version 4.8.8, Racket is version 6.10.1, and Chez Scheme is version 9.5.1. All time measurements use real time and we report the mean of 10 repeated measurements.

Measuring the Performance Lattice. Takikawa et al. [2016] 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 very many versions of a program that can be obtained by adding/removing static types. For languages with *coarse-grained gradual typing*, as in Takikawa et al. [2016], 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 Grift, 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.

For our experiments on partially typed programs, we randomly sample 900 configurations from across the spectrum of type annotations for each benchmark. In our experience, more samples for these benchmarks yields no new information, and makes interpreting the plots harder. Our sampling approach starts from a statically-typed program, creates a list of bins of how much type precision we want for the output program, creates a list of associations between source locations and type annotations, and shuffles the list to ensure randomness. The algorithm then proceeds to pick gradual versions of these types in such a way that the cumulative type precision of the resulting gradual types falls within the desirable bin. The types in each generated list is inserted back to the source program in place of the original type to create one sample. The algorithm loops until the desired number of samples have been generated from each bin.

4.2 The Runtime Cost of Space Efficiency

To evaluate the cost of maintaining space efficiency using coercions, we introduce another configuration of Grift, called *type-based casts*, which uses runtime type information in a straightforward (but naive) way to perform

[1] https://github.com/Gradual-Typing/benchmarks/tree/master/src

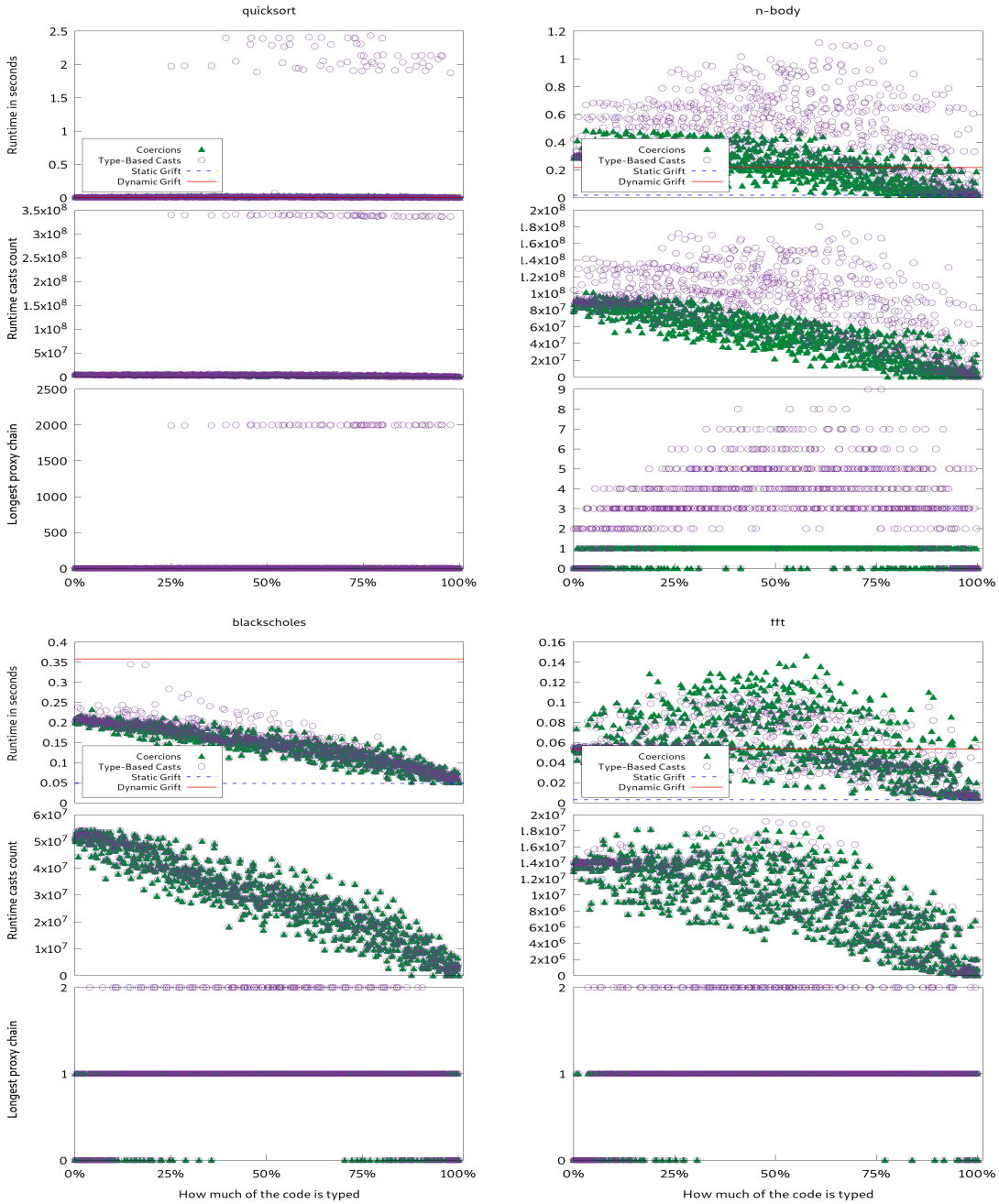


Fig. 6. We compare Grift with coercions to Grift with type-based casts (both use proxied references) across partially typed-programs to evaluate the cost of space-efficiency. The performance with coercions ranges from 0.91× to 13.65× that of Dynamic Grift. Coercions eliminate the catastrophic slowdowns in type-base. For quicksort, it is hard to see the data points for the *Coercions* configuration, but these data points the same data points that are label *Proxied* in Figure 7.

casts instead of coercions. The implementation is based on earlier work on gradual typing [Siek and Garcia 2012; Siek and Taha 2006], and described in detail by Kuhlenschmidt et al. [2018].

In Figure 6 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. For each benchmark 900 configurations are sampled (recall the discussion in Section 4.1).

In Figure 6 each figure has three plots and all of them 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. The y-axis is the absolute runtime in seconds in the first plot, and the number of runtime casts occurred at runtime in the second. In the third plot, it is the length of the longest chain of proxies accessed at runtime to perform a function application or a read or write operation on a reference/array. The line marked Dynamic Grift indicates the performance of Grift (using coercions and proxied references) on untyped code. That is, on code in which every type annotation is Dynand 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 4.4).

The quicksort benchmark elicits extreme space efficiency problems. In that benchmark, type-based casts exhibit catastrophic performance on some configurations. Recall from Section 2 that the accumulation of proxies changes the time complexity of quicksort from $O(n^2)$ to $O(n^3)$! This occurs when the array is cast on each recursive call, and each cast adds another proxy that induces overhead on subsequent reads and writes to the array. Indeed, the graph concerning the longest proxy chains for quicksort in Figure 6, shows that the configurations with catastrophic performance are the ones that accumulate proxy chains of length 2000.

On the other hand, the coercion-based approach successfully eliminates these catastrophic slowdowns in quicksort. The scale of this figure makes it hard to judge their performance in detail as the runtimes are so low relative to the type-based casts. The reader can look at the proxied data set in Figure 7 to zoom in on the same data. 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. Across all benchmarks the speedup of coercions over type-based cast is between 0.91× (the fft benchmark) and 13.65× (the quicksort benchmark).

> **Answer to research question (1):** On benchmarks without space-efficiency issues, we sometimes see a mild speedup and sometimes a mild slowdown. However, where space efficiency is needed, coercions eliminate the catastrophic slowdowns.

4.3 Monotonic versus Proxied References

In this section we investigate the differences in performance between monotonic and proxied references. We note that the semantics of monotonic references is different from proxied references. Both reference exist types within the same language. A third type of references can be switch between monotonic and proxied semantics at compile time. All benchmarks were written with this third type of reference, we refer to compiling with monotonic references as Monotonic Grift and with proxied references as Proxied Grift. Both versions use coercions (not type-based casts) as described in Section 3.

Figure 7 shows the runtime and number of casts for Monotonic versus Proxied Grift. For these results we do not report longest proxy chains because, for coercions the longest chain is either 0 or 1 as shown in Figure 6, and for monotonic there are no proxies.

For all the benchmarks, monotonic outperforms or matches proxied references. In cases where coercions compress casts to maintain a single proxy, as in quicksort and n-body, there can be significant speedups by using monotonic references. On the other hand, monotonic and proxied references have identical performance

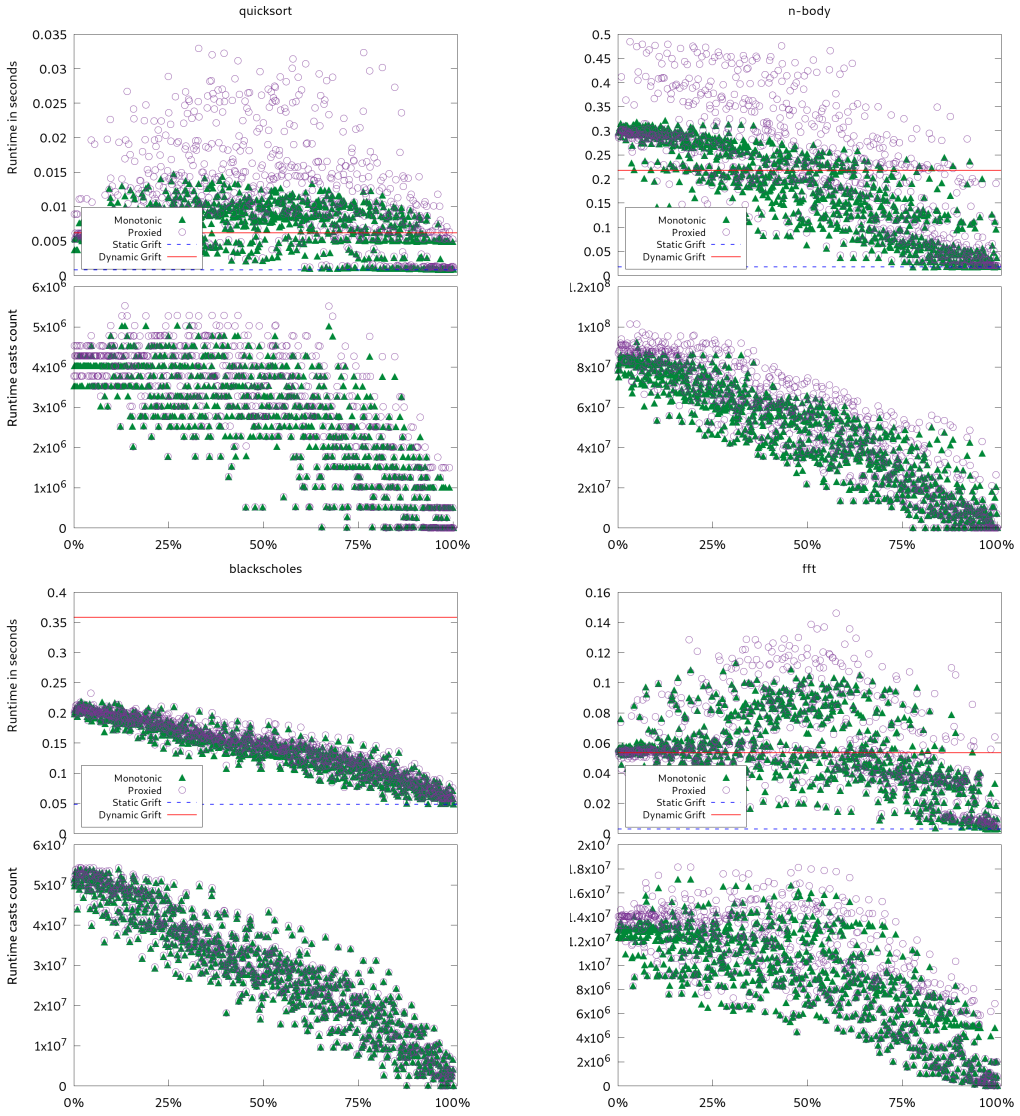


Fig. 7. Evaluation of Monotonic Grift and Proxied Grift on partially typed programs. The y-axes are runtime and number of casts. The x-axis varies the amount of type annotations in the program, from 0% to 100%. On the whole, monotonic references are more efficient than proxied references. The performance of Grift (especially monotonic) on partially typed code is quite reasonable: there are no catastrophic slowdowns and worst-case speedup with respect to Dynamic Grift is 0.42×.

on blackscholes. There is a correlation between runtime and the number of casts performed by each system. Dynamic Grift does significantly worse than any of partially-typed configurations on blackscholes, even the configuration that is 0% typed. Recall that for the Dynamic Grift experiments, we explicitly cast every constructed value to Dyn, whereas in the partially-typed configurations we only vary the pre-existing type annotations. In

blackscholes, this difference affects whether several vectors (initialized with constants) contain boxed versus unboxed values.

> **Answer to research question** (2)**:** Monotonic references are more efficient than proxied ones on partially typed programs.

4.4 Gradual Typing Overhead and Comparison

The purpose of this section is to answer research question 3, i.e., what is the overhead of gradual typing? Of course, to ultimately answer this question one would need to consider all possible implementations. So the actual question we answer is: what is the overhead of gradual typing in an ahead-of-time compiler that implements space-efficient coercions and monotonic references? To answer this question, it is important to isolate the overheads of gradual typing from other sources of overhead. Thus, we have implemented 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. We then compare (gradually typed) Grift with Static Grift.

Of course, one threat to validity is the possibility that the performance of Static Grift could be so poor that the overheads due to gradual typing are drowned out. To allay this fear, we include external comparisons to other programming language implementations, including statically typed languages such as OCaml, and dynamically typed languages such as Scheme and Racket. The upshot of this comparison is that the performance of Grift is in the same ballpark as these other implementations.

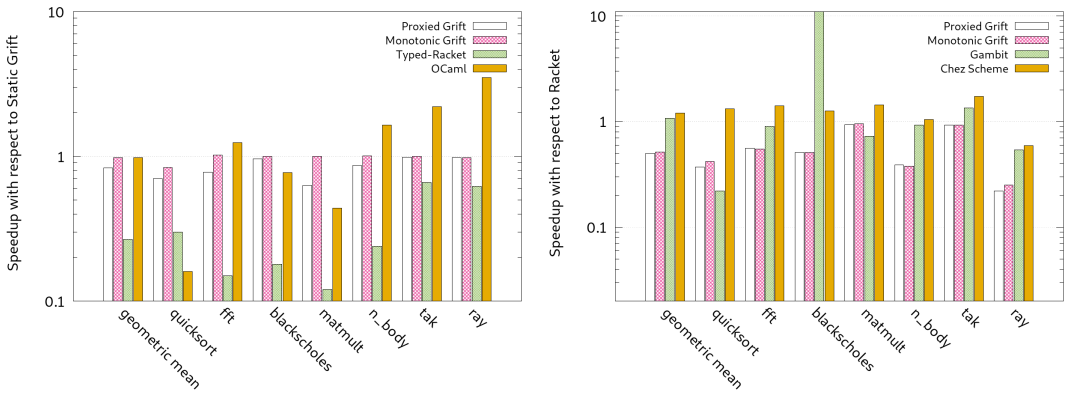
We attempt to mitigate some of the differences between languages and implementations. We use internal timing so that any differences in runtime initialization don't count towards runtime measurements. For Chez and Gambit we use the safe variants of fixnum and floating point specialized math operations, 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. We make no attempt to account for the difference in garbage collection between the languages. Instead, 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.

4.4.1 Evaluation on Statically Typed Programs. Figure 8a shows the results of evaluating the speedup of Proxied and Monotonic Grift with respect to Static Grift on statically typed versions of the benchmarks. We see that the performance of Monotonic Grift is no lower than 0.99× of Static Grift on all the benchmarks whereas the performance of Proxied Grift sometimes dips to 0.65× 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 (3 a):** the overhead of gradual typing for statically typed code is consistently low with monotonic references but sometimes high with proxied references.

4.4.2 Evaluation on Untyped Programs. Figure 8b shows the results of evaluating the speedup of Proxied and Monotonic Grift with respect to Racket on untyped versions of the benchmarks. For purposes of isolating causes of performance differences, it may be better to compare to a variant of Grift designed for only untyped program (analogous to Static Grift, but for dynamic languages), but we have not yet implemented such a variant. 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. 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 (3 b)** the overhead of gradual typing on untyped code is currently reasonable but there is still some improvement to be made.



(a) Statically Typed Programs

(b) Untyped Programs

Fig. 8. A comparison of the speedup on typed and untyped programs of Proxied Grift and Monotonic Grift. For typed programs, we measure speedup wrt. Static Grift and compare Grift with OCaml and Typed Racket. Monotonic Grift has consistently low overhead, while Proxied Grift occasionally shows overhead, but both are in the ballpark of OCaml and Typed Racket. For untyped programs, we measure speedup wrt. Racket and compare to Gambit and Chez Scheme. There is little difference between Proxied and Monotonic Grift on untyped code. Grift's performance is lower than Racket, Gambit, and Chez Scheme's.

4.4.3 Evaluation on Partially Typed Programs. We return to the results from Figure 7 to answer research question (3 c). As mentioned in the introduction the ideal performance model would be a linear model, where the dynamic typed program has "bad" performance, the statically typed program has "good" performance, and adding more types gradually improves the performance.

Of all the benchmarks only blackscholes behaves in the idealized model. The n-body benchmark comes pretty close for monotonic, but has some performance regressions in mostly untyped code. Coercions do get rid of catastrophic failures, but have unpredictable performance in the quicksort and n-body benchmarks which perform composition operations to maintain a single layer of proxies. In all benchmarks the number of casts tracks the ideal performance model, this suggests that if we could reduce the cost per cast we may be able to come closer to ideal performance.

> **Answer to research question (3 c):** the overhead of gradual typing on partially typed code is currently reasonable when considering the performance of monotonic references but there is still some improvement to be made for monotonic, and a lot of room for improvement for proxies.

5 CONCLUSION

We have presented Grift, a compiler for exploring implementations of gradually typed languages. In particular, we implement several technologies that enable gradual typing: type-based casts, space-efficient coercions, traditional proxied references, and monotonic references. Our experiments show that Grift with monotonic references is competitive with OCaml on statically typed code. For untyped code, Grift is on par with Scheme implementations using both proxied and monotonic implementations. On partially typed code, our experiments show that coercions eliminate the catastrophic slowdowns (changes to the time complexity) caused by space-inefficient casts. Furthermore, we see significant speedups (10×) as 60% or more of a program is annotated with types. Future work remains to improve the efficiency of coercions.

REFERENCES

John Allen. 1978. *Anatomy of LISP.* McGraw-Hill, Inc., New York, NY, USA.

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

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

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

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.

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.

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

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

M. Chevalier-Boisvert and M. Feeley. 2015. Interprocedural Type Specialization of JavaScript Programs Without Type Analysis. *ArXiv e-prints* (Nov. 2015). arXiv:cs.PL/1511.02956

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.

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.

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *ESOP*.

Marc Feeley. 2014. *Gambit-C: A portable implementation of Scheme.* Technical Report v4.7.2. Universite de Montreal.

Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. Collapsible Contracts: Fixing a Pathology of Gradual Typing. (2018). to appear.

R. B. Findler and M. Felleisen. 2002a. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*. 48–59.

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

Ronald Garcia. 2013. Calculating Threesomes, with Blame. In *ICFP '13: Proceedings of the International Conference on Functional Programming*.

P. Graham. 1995. *ANSI Common Lisp.* Prentice Hall.

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.

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

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2010. Contracts Made Manifest. In *Principles of Programming Languages (POPL) 2010*.

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.

Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. 2007. Relationally-Parametric Polymorphic Contracts. In *Dynamic Languages Symposium*.

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

Anders Hejlsberg. 2012. Introducing TypeScript. Microsoft Channel 9 Blog. (2012).

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

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

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

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2018. Efficient Gradual Typing. (February 2018). arXiv:1802.06375 https://arxiv.org/abs/1802.06375

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)*.

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*.

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

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. 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

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

Jeremy G. Siek. 2008. Space-Efficient Blame Tracking for Gradual Types. (April 2008).

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

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

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

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

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

Jeremy G. Siek and Philip Wadler. 2009. Threesomes, with and without blame. In *Proceedings for the 1st workshop on Script to Program Evolution (STOP '09)*. ACM, New York, NY, USA, 34–46. https://doi.org/10.1145/1570506.1570511

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)*.

Asumu Takikawa, Daniel Feltey, Earl Dean, Matthew Flatt, Robert Bruce Findler, Sam Tobin-Hochstadt, and Matthias Felleisen. 2015. Towards Practical Gradual Typing. In *European Conference on Object-Oriented Programming (LIPICS)*. Dagstuhl Publishing.

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.

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

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

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

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

A VALUES, MACROS, AND COMPOSE

Figure 9 lists the C structs use to represent values. Figure 10 lists the macros for manipulating values. Figure 11 show the code for compose runtime function which directly follows the equations for compose which can be found in Kuhlenschmidt et al. [2018].

```
/* Types */
typedef type* ref_type;
typedef struct {int64_t arity; type ret; type args[]} fun_type;
typedef union {int64_t atm; ref_type* ref; fun_type* fun;} type;
/* Coercions */
typedef struct {type to; blame info;} project_crcn;
typedef type* inject_crcn;
typedef struct {crcn fst; crcn snd;} seq_crcn;
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} pref_crcn;
typedef struct {snd_tag second_tag; crcn rtti} mref_crcn;
typedef struct {char* lbl} fail_crcn;
#define ID NULL
/* Values */
#ifdef TYPE_BASED_CASTS
  typedef struct {obj* ref; type source; type target blame info;} ref_proxy;
  typedef struct {void* code;  (obj)(*caster)(obj, type, type, blame);  obj[]; } closure;
#elseif COERCIONS
  typedef struct {obj* ref; crcn cast;} ref_proxy
  typedef struct {void* code; (obj)(*caster)(obj, type, type, blame);
                  union {crcn  coerce; obj[] fvars;} } closure;
#endif
typedef struct {obj value; type source} nonatomic_dyn;
typedef union {int64_t atomic; nonatomic_dyn*} dynamic;
typedef union {int64_t fixnum; double  flonum; closure* clos; dynamic dyn} obj;
```

Fig. 9. Value representations

```
/* All allocated values have 3 bits that can be used for tagging */
#define TAG(value) (((int64_t)value)&0b111)
#define UNTAG_INT(value) (((int64_t)value)&~0b111)
#define TAG_INT(value, tag)   (((int64_t)value)|tag)
/* Macros that manipulate types */
#define HEAD(type) (TAG(type))
#define ARITY(type) (((fun_type)UNTAG_INT(type))->arity)
#define REF_TYPE(type) (*((ref_type)UNTAG_INT(type)))
/* Macros that manipulate values in the obj union */
#define UNTAG_REF(ref) ((obj*)UNTAG_INT(ref))
#ifdef TYPE_BASED_CASTS
#define UNTAG_FUN(fun) ((closure*)(fun)))
#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)
#elseif COERCIONS
#define UNTAG_FUN(fun) ((closure*)UNTAG_INT(fun))
#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
/* Macros that manipulate values in the dynamic union */
#define UNTAG_NONATOMIC(value) ((nonatomic_dyn)UNTAG_INT(value))
#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)
/* Macros that manipulate types in the crcn union */
#define UNTAG_2ND(c) ((struct {snd_tag second_tag;}*)UNTAG_INT(c))
/* 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)))
```

Fig. 10. Macros for manipulating values

```
crcn compose(crcn fst, crcn snd) {
  if (fst == ID) { 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) {
      return MK_SEQ(s1->fst, compose(s1->snd, snd)); }
    else if (TAG(snd) == FAIL_TAG) { 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);
      return compose(compose(seq->fst, c), s2->snd);
    }
  } else if (TAG(snd) == SEQUENCE_TAG) {
    if (TAG(fst) == FAIL) { return fst; }
    else {
      crcn c = compose(fst, s2->fst);
      return MK_SEQ(c, UNTAG_SEQ(seq)->snd); }
  } else if (TAG(snd) == FAIL) {
      return TAG(fst) == FAIL ? fst : snd; }
  } else if (TAG(fst) == HAS_2ND_TAG) {
    snd_tag tag = UNTAG_2ND(fst)->second_tag;
    if (tag == FUN_COERCION_TAG) {
      return compose_fun(fst, snd); }
    else if (tag == 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 { raise_blame(UNTAG_FAIL(fst)->lbl); }
}
```

Fig. 11. The compose function for normalizing coercions.