Sound and Efficient Fine-Grained Gradual Typing via Contract Verification

TEMUR SAIDKHODJAEV, University of Maryland, USA

Gradual typing allows mixture of static and dynamic typing in the same program. To provide the usual static typing guarantees, runtime type checks have to be inserted at the boundaries between typed and untyped code. Unfortunately, these checks incur significant performance overhead. Recently, soft contract verification has been applied with great success in coarse-grained gradual typing. The key idea of this approach is that untyped portions of the program can be statically shown to maintain the type system invariants, so the runtime checks can be removed. This thesis applies the same approach in a fine-grained setting.

I implement the contract verification optimization in a fine-grained gradual language, and evaluate it on a few existing benchmarks. The optimization removes all runtime type checks in the benchmark programs, achieving speedups ranging from $7\times$ to $140\times$ compared to the best-performing configuration of the language.

1 INTRODUCTION

Recently, gradual type systems have gained adoption [Chaudhuri et al. 2017; Microsoft Corp. 2014; Stripe Inc. 2019; Tobin-Hochstadt and Felleisen 2008]. Gradual typing allows a mixture of static and dynamic typing within the same program, making it possible for programmers to decide which approach they want to use in a particular situation. The hope is that fully typed portions of the program provide all the benefits of static typing, while dynamic portions are more flexible and can use dynamic idioms.

The question arises, how do statically typed and dynamically typed parts of the program work together? One solution is *higher-order contracts* [Findler and Felleisen 2002], which are inserted at the borders between typed and untyped code with the purpose of protecting the invariants of static regions. *Sound* gradual typing protects all channels of communication between untyped and typed code, ensuring that the type annotations are reliable.

However, due to the inherent lack of type information about the dynamically typed parts of the code, the inserted contracts have to be executed at runtime, which incurs some overhead. Unfortunately, recent studies [Greenman et al. 2019; Takikawa et al. 2016] show that this runtime overhead is prohibitively large. Depending on which parts of the program are statically typed, the slowdowns can reach over $20\times$. To avoid such drastic performance issues, many gradually typed languages choose not to perform some or all runtime type checks, which results in the loss of soundness. In practical terms, this means losing out on error reporting and type-based optimizations that full gradual typing provides [Greenman and Felleisen 2018].

Recently, Moy et al. [2021] used higher-order symbolic execution to verify some or all of the runtime type checks, allowing the compiler to not insert them at all. The core idea of their approach is that "*dynamic contracts are statically useful*", so the authors use an existing higher-order symbolic executor for Racket called SCV [Nguyễn et al. 2018] to try and verify as many of those contracts as possible. The results are impressive, as almost all contracts are verified away in the set of gradual typing benchmarks [Greenman et al. 2019], leaving practically no performance overhead. The analysis is also modular, in the sense that each module can be analyzed separately, and any verification failures don't affect the verification process in other modules.

In their work, Moy et al. [2021] concentrate on Typed Racket, which implements a flavor of gradual typing called *coarse-grained* gradual typing. In a coarse-grained system, a single module must either be fully typed or fully untyped. This is opposed to *fine-grained* gradual typing, where

Author's address: Temur Saidkhodjaev, University of Maryland, College Park, MD, USA.

a programmer has access to an unknown type `Dyn` that can be used to designate untyped parts of the code.

This work develops the ideas of Moy et al. [2021] by applying the same ideas in the fine-grained setting, in particular in Grift [Kuhlenschmidt et al. 2019], which implements sound fine-grained gradual typing and attempts to reduce performance overheads using coercions. This presents a different set of challenges compared to the original work on Typed Racket. First, Grift is a research system designed from scratch to be gradually typed, so the number of features is limited. Second, since Grift is fine-grained, the same notion of modularity cannot be applied. Third, Grift is a different language from Racket, so using the Racket-oriented symbolic executor requires a semantics-preserving translation from Grift to Racket.

Contributions. This thesis contributes:

- a technique for optimizing fine-grained gradually typed programs by verifying boundary contracts generated by the gradual type system;
- SCV-Grift, a tool implementing this technique, integrating Grift and an existing contract verification system;
- and an evaluation on a few preexisting benchmarks showing the effectiveness of this approach.

On all of the benchmarks supported by SCV-Grift, 100% of the contracts were verified, thus resulting in no slowdowns. This is a significant improvement compared to Grift optimized with coercions, which removes catastrophic overheads on the order of $10\times$, but still incurs overheads on the order of $1\times$. The rest of this report explains the core idea of the approach, describes the implementation, provides a performance evaluation, and discusses related work.

2 OPTIMIZED FINE-GRAINED GRADUAL TYPING EXPLAINED

This section uses a gradually typed program to explain the ideas behind the contract verification approach to gradual typing optimization.

2.1 Example Description

Figure 1a shows CPS-EVEN-ODD, a synthetic gradual typing benchmark that checks whether a given integer is even or odd using mutual recursion and continuation passing style.

While this program is not realistic, it does exhibit pathological space consumption when runtime casts are implemented naively [Herman et al. 2007]. You can see the intermediate representation of the same program in Figure 1b. This representation is called the Grift cast calculus, which is Grift's intermediate representation with explicit casts. The cast `(cast k (Bool -> Bool) (Dyn -> Bool) "l7")` causes trouble. Normally, this higher-order cast is implemented by wrapping a proxy around the value `k`, which instead of just calling `k`, first checks the input against the expected type, then calls `k`, and then checks the output against the expected type. In this example, every time `k` is passed to the recursive call, it is wrapped by a new proxy, so by the time `k` is called, it is wrapped by $O(n)$ proxies, where n is the input number. Since each proxy requires allocation, this results in $O(n)$ space consumption; without casts tail call elimination makes space consumption $O(1)$.

CPS-EVEN-ODD is a great example of pathological space consumption in gradual typing, but as Kuhlenschmidt et al. [2019] show, some pathological slowdowns happen due to the same repeated wrapping of proxies around a function. In fact, in this example, when `k` is finally called, $O(n)$ type checks are executed, which doesn't result in an asymptotic slowdown, but is still significant.

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

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

(define (empty-k [k : Dyn]) : Dyn
  k)

(define (run-benchmark) : Unit
  (print-bool
    (even? (read-int) empty-k)))

(run-benchmark)
```

(a) Grift

```
(define (even? n k)
  (cast
    (if (= (cast n Dyn Int "l1") 0)
      (k (cast #t Bool Dyn "l2"))
      (odd? (- (cast n Dyn Int "l3") 1)
            (cast k
              (Dyn -> Bool)
              (Bool -> Bool) "l4"))
    Bool Dyn "l5"))

(define (odd? n k)
  (if (= n 0)
    (k #f)
    (cast
      (even? (cast (- n 1) Int Dyn "l6")
             (cast k
              (Bool -> Bool)
              (Dyn -> Bool) "l7"))
      Dyn Bool "l8")))

(define (empty-k k)
  k)

(define (run-benchmark)
  (print-bool (cast
                (even?
                  (cast (read-int)
                    Int Dyn "l9")
                  (cast empty-k
                    (Dyn -> Dyn)
                    (Dyn -> Bool) "l10"))
               Dyn Bool "l11")))

(run-benchmark)
```

(b) Grift Cast Calculus

Fig. 1. CPS-EVEN-ODD original code and stage 1 of the optimization.

2.2 Optimization Overview

The optimization happens in three stages:

- Cast insertion. This stage involves translating a gradually typed program into the Grift cast calculus.
- Translation to Racket with contracts. At this stage, the untyped program with casts is translated to Racket with contracts while preserving the semantics of the program.
- Contract verification and erasure. The symbolic executor SCV is run on the translated Racket code, and the contracts whose assumptions always hold are marked safe, while those that possibly fail are marked unsafe. The casts corresponding to safe contracts are then be removed, as they always hold. The optimized program is then passed to the rest of the compilation pipeline.

2.3 Cast Insertion

In a sound gradually typed language, the compiler sometimes has a pass that produces an intermediate language with explicit casts [Herman et al. 2007; Siek and Taha 2006], and Grift is no exception [Kuhlenschmidt et al. 2019]. Since the compiler handles this process, it is only needed to intercept the intermediate cast calculus.

For CPS-EVEN-ODD example, you can find the cast calculus representation in Figure 1b. The casts are represented using a `(cast e t1 t2 lbl)` form, where `e` is an expression, `t1`, `t2` are types and `lbl` is an auxiliary label. In the Grift compiler, this intermediate representation is in the form of an abstract syntax tree.

2.4 Translation to Racket with Contracts

The intermediate cast calculus is now translated into Racket with contracts. I will call the language from which the translation happens the *host* language, which in this case is Grift. The translation process is highly dependent on the feature set of the host language, and might be very difficult if Racket's feature set is vastly different from the host language. For Grift, the translation is straightforward once the mapping between Grift types and Racket types is established.

The Racket translation of CPS-EVEN-ODD example can be found in Figure 2a. Primitives, such as `read-int` or `print-bool` that are not present in the Racket standard library are implemented in Racket and guarded with the appropriate contracts. Casts are translated to contracts, and the auxiliary labels are used to establish a correspondence between casts and contracts, which is later used to erase verified casts.

2.5 Contract Verification and Erasure

In this step, the symbolic executor SCV is run on the translated Racket code, and any given contract is either proved safe or marked as unsafe. It is important to note that when a contract is marked as unsafe, it does not mean that it is guaranteed to fail, but rather that SCV was unable to prove its safety. Since the host to Racket translation preserves the semantics of the program and SCV is sound [Nguyễn et al. 2018], casts corresponding to the safe contracts can be erased. The resulting cast calculus representation is now proven to be well-typed, which opens the door to type-based optimizations.

Contracts highlighted green in Figure 2a are marked safe by SCV, so the corresponding casts are removed from the intermediate representation and will not be found in Figure 2b. In this example, all casts are verified away, so there are no `cast` forms in Figure 2b. Generally, it is difficult to construct a well-typed Grift program where any cast is not verified by SCV due to Grift's limited type system and small feature set.

2.6 Evaluation

The contract verification approach is clearly successful at optimizing the CPS-EVEN-ODD example. All contracts are verified and erased, which both removes the increased space consumption and eliminates slowdowns. Running this benchmark program five times in SCV-Grift shows an average runtime of 34ms, while Kuhlenschmidt et al. [2019] report runtime of more than a second on the same input.[1]

[1] I was unable to reproduce their results for this example for a more fair comparison.

```
#lang racket
(require "grift-primitives.rkt")

(define (even? n k)
  (contract any/c
    (if (=
          (contract exact-integer?
            n 'l1 'l1)
          0)
      (k (contract any/c #t 'l2 'l2))
      (odd?
        (-
          (contract exact-integer?
            n 'l3 'l3)
          1)
        (contract
          (-> boolean? boolean?)
          k 'l4 'l4)))
    'l5 'l5))

(define (odd? n k)
  (if (= n 0)
    (k #f)
    (contract boolean?
      (even?
        (contract any/c
          (- n 1) 'l6 'l6)
        (contract (-> any/c boolean?)
          k 'l7 'l7))
      'l8 'l8)))

(define (empty-k k)
  k)

(define (run-benchmark)
  (print-bool
    (contract boolean?
      (even?
        (contract any/c
          (read-int) 'l9 'l9)
        (contract (-> any/c boolean?)
          empty-k 'l10 'l10))
      'l11 'l11)))

(run-benchmark)
```

(a) Racket, Safe Contracts are in Green

```
(define (even? n k)
  (if (= n 0)
      (k #t)
      (odd? (- n 1) k)))

(define (odd? n k)
  (if (= n 0)
      (k #f)
      (even? (- n 1) k)))

(define (empty-k k)
  k)

(define (run-benchmark)
  (print-bool
    (even? (read-int) empty-k)))

(run-benchmark)
```

(b) Optimized Grift Cast Calculus

Fig. 2. CPS-EVEN-ODD in stages 2 and 3 of the optimization.

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

Fig. 3. The subset of Grift syntax supported by current version of the optimization.

3 IMPLEMENTATION

SCV-Grift is implemented as an optional pass in the Grift compiler.[2] The Grift compilation pipeline consists of four major passes: reduction to the cast calculus, imposing of cast semantics, conversion of expression-oriented language to a statement-oriented language, and C code generation. Each of these passes may consist of multiple other passes, but those are irrelevant to the discussion. SCV-Grift intercepts the cast calculus representation between the first and the second pass, performs the optimization by removing verified casts from the AST, and feeds the result to the next pass.

Figure 3 [Kuhlenschmidt et al. 2019] shows the syntax of the subset of the Grift language that is supported by SCV-Grift. Grift supports a rich set of features on top of the standard gradually typed lambda calculus [Siek and Taha 2006], such as tuples, mutable references, mutable arrays, and equirecursive types [Kuhlenschmidt et al. 2019]. The only feature that it currently not supported by SCV-Grift is equirecursive types, and it is thus not included in Figure 3.

As described before, the optimization consists of three major steps: cast insertion, translation to Racket with contracts, contract verification and erasure. Since Grift already performs cast insertion, this step only requires intercepting the intermediate representation generated by Grift. The next two subsections discuss the remaining two steps.

[2]The modified Grift compiler can be found at https://github.com/Temurson/Grift/tree/scv-cr-fine-grained.

3.1 Translation to Racket with Contracts

All Grift features are directly supported by Racket, albeit in an untyped setting. Thus the translation itself is straightforward, and the only challenge is translating the casts correctly. The ranges of all the ground types in Grift match those in Racket, so the only concern is to identify the contracts describing those types exactly. Integers are checked by the contract `exact-integer?`, and floats are checked by `flonum?`, while all other ground types are checked by the same contracts as in Racket. Higher-order casts are translated to contracts verbatim. Tuples are translated to Racket lists with contracts checking the size of the list and all types of all the elements. Mutable references are translated to Racket's boxes, and mutable arrays (vectors) are translated to Racket's vectors.

All casts to type `Dyn` are translated as `any/c`, meaning that they always hold. This is not entirely correct, as Grift maintains two representations of values, boxed and unboxed, and casts to `Dyn` possibly change the representation of the value to a boxed one, so removing this cast is dangerous. However, this is only a concern if there is an unboxing cast that is left unverified, which was never the case in any of the benchmark programs. The solution to this is incorporating the semantics of boxing and unboxing into the translation, which will be added to SCV-Grift in the future.

Most language forms in Grift are present in Racket, so they are translated verbatim. The only caveat is that `tuple-proj` should be translated to named list accessors, such as `third`, in order to increase analysis precision and correctly capture the semantics of `tuple-proj`, whose index argument is always a literal integer. The `repeat` form is just a variation of Racket's for loops, and is translated as such. The `time` form, being an auxiliary construct for benchmarking, is not translated at all, so only its underlying expression is left after the translation.

All the primitive operations supported by Grift are translated verbatim, but all those operations are then reimplemented in Racket and imported as a library in the translated file. Most operations have direct counterparts in Racket, so their implementation is straightforward, but they need to be guarded by appropriate contracts to maintain the type signatures of the original Grift primitives.

When casts are translated, each cast is assigned a unique label, and that label is then assigned a blame label that is attached to the corresponding contract. This establishes a one-to-one relationship between casts and contracts explicitly inserted during the translation, which makes identification of the unsafe contracts easier in the next step.

3.2 Contract Verification and Erasure

On the final stage, the translated Racket code is saved as a file, all necessary imports are added, and SCV is run on the translation. SCV here is treated as a black box that determnies which contracts are unsafe. Such contracts cannot be removed, but all the other ones are considered safe, so they can be safely erased. SCV may report possible contract failures when using library functions, or failures that are not related to the contracts inserted during the translation. To weed out the irrelevant contracts, the unique labels mentioned before are used.

In the verification process, SCV is run not only on the primary translation file, but also on the library file containing the Grift primitives reimplemented in Racket. Those primitives are often guarded by contracts that simply do not hold in Racket, but are necessary to preserve the Grift semantics. For example, `read-int` is translated as `read` guarded by the contract `(-> exact-integer?)`, which SCV rightfully detects as a possible failure. All such failures are ignored, and do not affect the overall verification process.

After all the safe contracts are erased, the optimized cast calculus AST is passed on to the next compilation pass, and the compilation process proceeds normally.

4 EVALUATION

The empirical claim of my thesis is that SCV-Grift removes almost all performance overhead of gradual typing. This claim is supported by an evaluation of a set of benchmarks used by Grift implementers and conducted according to the standard gradual typing benchmarking methodology [Takikawa et al. 2016].

4.1 Benchmark Programs

The benchmark programs for the evaluation are developed by Kuhlenschmidt et al. [2019], who in turn compiled some of those benchmarks from the Scheme benchmark suite (R6RS) used to evaluate the Larceny [Hansen and Clinger 2002] and Gambit [Feeley 2014], the PARSEC benchmarks [Bienia et al. 2008], the Computer Language Benchmarks Game, and the Gradual Typing Performance Benchmarks [Greenman et al. 2019]. Kuhlenschmidt et al. [2019] use eight benchmark programs, but this report only presents four of those. One was not used because SCV-Grift does not yet support Grift's equirecursive types, and I was unable to reproduce Grift's performance results for the other three.

The measurements are made in comparison to Grift with coercions, which is the best-performing configuration of the Grift compiler.

Each benchmark is a single-file fully-typed Grift program, for which various gradually typed configurations are generated. This is done in accordance with the gradual typing benchmarking methodology described by Takikawa et al. [2016], with some modifications by Greenman and Migeed [2017].

The original methodology by Takikawa et al. [2016] starts with a fully typed program, then, depending on the granularity of the gradual system, the smallest portions of code where types can be removed are identified. In case of a fine-grained system like Grift, these are the type annotations. Then, for a given program, all possible gradually typed configurations are generated by replacing an explicit type by the type `Dyn`. If there are n type annotations in a given program, this results in a lattice of size 2^n containing all possible partially typed configurations, starting from a fully typed program, and ending with a fully untyped program. These programs are then run, and the results show how well a gradual system performs in partially typed configurations. However, the number of configuration in the fine-grained setting is too large to measure them all, so Greenman and Migeed [2017] propose to sample a linear number of configurations with respect to the program size, and show that this approach estimates the performance of the entire lattice well.

The benchmark programs are:

- RAY. Ray tracing a scene, 20 iterations. This is a test of floating point arithmetic.
- QUICKSORT. The textbook quicksort algorithm run on a worst-case input with integer arrays of size 1000.
- MATMULT. The naive matrix multiplication algorithm consisting of three nested loops, with integer elements on matrix sizes 400×400.
- ARRAY. This benchmark allocates and destructively initializes two one-dimensional arrays of size 500, repeated 100000 times.

4.2 Experimental Setup

The experiments are run in the Ubuntu virtual machine image Kuhlenschmidt et al. [2019] provide as an artifact for their paper, which was allocated 4 cores and 8 GB of RAM from my personal machine with a 12-core Intel Core i7 CPU @ 2.60GHz processor and 16 GB of RAM running Ubuntu 20.04. All the software versions match those in the Grift artifact, the only difference being the

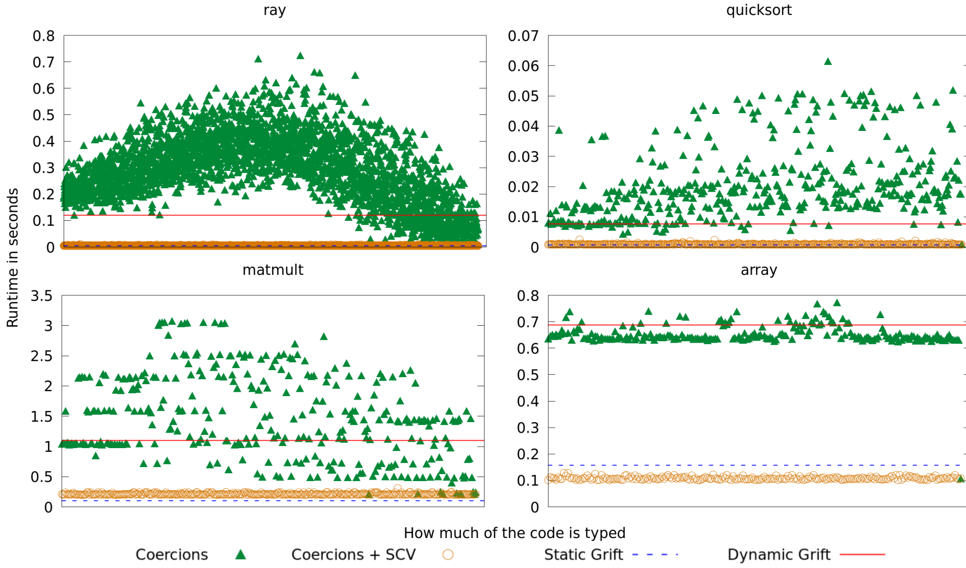

Fig. 4. Runtimes for the four selected benchmarks.

installation of SCV and addition of SCV-Grift to the Grift compiler. The old version of the Grift compiler was used to provide a more fair comparison.

Each configuration is run three times and the average runtime is used. Baseline measures of fully static and fully dynamic Grift are added to the plots for comparison.

4.3 Results

Figure 4 shows the runtime comparison of Grift with coercions versus Grift with coercions and SCV-Grift. Since SCV-Grift verifies and erases all contracts in all of the configurations, the runtimes for SCV-Grift are consistently low regardless of the gradually typed configuration. In contrast, the runtime performance of Grift with coercions is highly dependent upon the configuration. In particular, maximum slowdowns in each benchmark compared to SCV-Grift are: RAY shows $140\times$ slowdown, QUICKSORT shows $65\times$ slowdown, MATMULT shows $15\times$ slowdown, and ARRAY shows $7\times$ slowdown.

4.4 Limitations and Future Work

Contract verification seems to be a promising approach in reducing performance overheads of gradual typing, as shown by Moy et al. [2021], and this work reinforces their result, now applied in a fine-grained setting. However, the current version of SCV-Grift is incomplete, as it does not support equirecursive types. Grift also uses a separate representation for values of ground types when they are cast to Dyn, which is not yet accounted for in SCV-Grift. This problem can be solved by incorporating the semantics of different representations in the translation process. Supporting more benchmarks will also improve the evaluation of this work.

I have to note that SCV-Grift is ultimately limited by the feature set of SCV, although that was not a problem in Grift, as Grift's feature set is rather limited itself. Supporting other gradually typed languages might be more challenging depending on how much their feature set differs from

Racket's. Currently, this report does not provide a formal proof that the Grift to Racket translation is correct, which is subject to future work.

5 RELATED WORK

Shortly after the term *gradual typing* was introduced by Siek and Taha [2006], researchers and implementers started to realize that runtime checks could impact performance. This was first noted by Herman et al. [2010], who showed that the accumulation of runtime checks can result in asymptotically worse space consumption, which was later addressed by a number of authors [Feltey et al. 2018; Greenberg 2016; Siek and Wadler 2010; Tsuda et al. 2020]. However, the ultimate demonstration of the problems with gradual typing performance was done by Takikawa, Greenman, and their collaborators, who also designed a methodology for evaluating performance of gradual languages, and a suite of gradually typed benchmarks [Greenman et al. 2019; Takikawa et al. 2015, 2016].

After this, the work on optimizing gradual typing proceeded in three major directions. First, there was a significant effort to improve the runtime enforcement mechanisms of gradual typing, which manifested in improvements to the underlying virtual machines [Bauman et al. 2017], and more efficient compilation of contracts [Feltey et al. 2018; Kuhlenschmidt et al. 2019]. Second, some work focused on restricting the language features to ensure both soundness and acceptable performance, which took form of restricting dynamic checks to use nominal types [Bierman et al. 2010; Muehlboeck and Tate 2017], and limiting the types of values to flow across untyped boundaries [Google Inc. 2018; Richards et al. 2017, 2015; Swamy et al. 2014; Wrigstad et al. 2010]. Third, the most common approach in industrially used gradual systems, such as TypeScript, Hack, Flow, Sorbet, MyPy, and others, was to omit some or all runtime checks, resulting in loss of soundness.

The most recent approach to optimizing gradual typing is using static analysis to eliminate the runtime checks entirely. This is the approach of Moy et al. [2021], who used a symbolic execution-based contract verifier SCV [Nguyễn et al. 2018] to verify type-checking contracts. A similar approach is taken by Vitousek et al. [2019] in their work on Reticulated Python, a gradual type system for the Python language. They designed a type inference algorithm that uses constraint solving, and applied it to their gradual system, achieving very low performance overheads when running their programs with PyPy, a tracing JIT compiler. However, in an effort to support real-world Python programs, they modified the cast insertion algorithm and relaxed the notion of soundness [Vitousek et al. 2017], which resulted in worse error reporting.

This report confirms the results of Moy et al. [2021] that sound gradual typing can be achieved without giving up language features or performance. Compared to all of the above approaches, SCV-Grift performs better or equally well on the supported benchmarks.

6 CONCLUSION

Ever since Takikawa et al. [2016] declared sound gradual typing to be dead because of the catastrophic performance overhead, there have been multiple attempts to "revive" it. Most of these attempts sought to improve the technology behind the runtime enforcement of gradual typing. While specific causes of catastrophic slowdowns have been eliminated in previous work, none have achieved acceptable performance in all situations.

That was until Moy et al. [2021] tried a different approach: they use contract verification to eliminate the need for runtime enforcement of gradual typing where possible. This approach proved to be successful, as nearly all overhead of gradual typing was eliminated in Typed Racket. The work of Moy et al. [2021] addressed the coarse-grained gradual typing, and this thesis

applied the same ideas to Grift [Kuhlenschmidt et al. 2019], a fine-grained gradually typed language.

I show that the contract verification approach to eliminating gradual typing overhead works in a fine-grained setting, by implementing a tool SCV-Grift that applies a Racket contract verifier SCV [Nguyễn et al. 2018] to a translated Grift program, and evaluating its performance on a few preexisting gradual typing benchmarks. All contracts are verified away in every benchmark, and resulting programs outperform the original Grift programs by a significant amount, removing all overhead of gradual typing. These results suggest that static analysis is a promising approach to reducing performance overhead of gradual typing.

More work is needed to confirm that the contract verification approach is capable of solving the gradual typing performance issues. In particular, both my work and the work of Moy et al. [2021] rely on a Racket symbolic executor SCV to verify contracts, but SCV does not yet support object-oriented language features. Moreover, applying the same approach in other gradually typed languages might not be viable without developing language-specific symbolic execution tools.

ACKNOWLEDGMENTS

I am grateful to the Corpse Reviver: Sound and Efficient Gradual Typing via Contract Verification authors Cameron Moy, Phúc C. Nguyễn, Sam Tobin-Hochstadt and David Van Horn, who is also my undergraduate research advisor, for continuous support and advice on this project. Thanks to Andre Kuhlenschmidt for support in understanding the Grift compiler. Thanks to University of Maryland Computer Science Honors Program Chairs Dave Levin and Leilani Battle for enabling my passion for research.

REFERENCES

Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. 2017. Sound Gradual Typing: Only Mostly Dead. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. DOI: http://dx.doi.org/10.1145/3133878

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. Association for Computing Machinery, New York, NY, USA, 72–81. DOI: http://dx.doi.org/10.1145/1454115.1454128

Gavin Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding Dynamic Types to C#. In *European Conference on Object-Oriented Programming (ECOOP)*. DOI: http://dx.doi.org/10.1007/978-3-642-14107-2_5

Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and Precise Type Checking for JavaScript. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. DOI: http://dx.doi.org/10.1145/3133872

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

Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. Collapsible Contracts: Fixing a Pathology of Gradual Typing. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. DOI: http://dx.doi.org/10.1145/3276503

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *International Conference on Functional Programming (ICFP)*. DOI: http://dx.doi.org/10.1145/581478.581484

Google Inc. 2018. Dart. (2018). https://dart.dev/

Michael Greenberg. 2016. Space-Efficient Latent Contracts. In *Trends in Functional Programming (TFP)*. DOI: http://dx.doi.org/10.1007/978-3-030-14805-8_1

Ben Greenman and Matthias Felleisen. 2018. A Spectrum of Type Soundness and Performance. In *International Conference on Functional Programming (ICFP)*. DOI: http://dx.doi.org/10.1145/3236766

Ben Greenman and Zeina Migeed. 2017. On the Cost of Type-Tag Soundness. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '18)*. Association for Computing Machinery, New York, NY, USA, 30–39. DOI: http://dx.doi.org/10.1145/3162066

Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. 2019. How to Evaluate the Performance of Gradual Typing Systems. *Journal of Functional Programming (JFP)* (2019). DOI: http://dx.doi.org/10.1017/S0956796818000217

Lars T. Hansen and William D. Clinger. 2002. An Experimental Study of Renewal-Older-First Garbage Collection. *SIGPLAN Not.* 37, 9 (Sept. 2002), 247–258. DOI: http://dx.doi.org/10.1145/583852.581502

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

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-Efficient Gradual Typing. *Higher-Order and Symbolic Computation* (2010). DOI: http://dx.doi.org/10.1007/s10990-011-9066-z

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. Toward Efficient Gradual Typing for Structural Types via Coercions. In *Programming Language Design and Implementation (PLDI)*. DOI: http://dx.doi.org/10.1145/3314221.3314627

Microsoft Corp. 2014. TypeScript Language Specification. (2014). http://www.typescriptlang.org

Cameron Moy, Phúc C. Nguyễn, Sam Tobin-Hochstadt, and David Van Horn. 2021. Corpse Reviver: Sound and Efficient Gradual Typing via Contract Verification. 5, POPL, Article 53 (Jan. 2021), 28 pages. DOI: http://dx.doi.org/10.1145/3434334

Fabian Muehlboeck and Ross Tate. 2017. Sound Gradual Typing is Nominally Alive and Well. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. DOI: http://dx.doi.org/10.1145/3133880

Phúc C. Nguyễn, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2018. Soft Contract Verification for Higher-Order Stateful Programs. In *Principles of Programming Languages (POPL)*. DOI: http://dx.doi.org/10.1145/3158139

Gregor Richards, Ellen Arteca, and Alexi Turcotte. 2017. The VM Already Knew That: Leveraging Compile-Time Knowledge to Optimize Gradual Typing. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. DOI: http://dx.doi.org/10.1145/3133879

Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete Types for TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*. DOI: http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.76

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*.

Jeremy G. Siek and Philip Wadler. 2010. Threesomes, with and without Blame. In *Principles of Programming Languages (POPL)*. DOI:http://dx.doi.org/10.1145/1706299.1706342

Stripe Inc. 2019. Sorbet. (2019). https://sorbet.org/

Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. 2014. Gradual Typing Embedded Securely in JavaScript. In *Principles of Programming Languages (POPL)*. DOI:http://dx.doi.org/10.1145/2535838.2535889

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 (ECOOP)*. DOI:http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.4

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *Principles of Programming Languages (POPL)*. DOI:http://dx.doi.org/10.1145/2837614.2837630

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Principles of Programming Languages (POPL)*. DOI:http://dx.doi.org/10.1145/1328438.1328486

Yuya Tsuda, Atsushi Igarashi, and Tomoya Tabuchi. 2020. Space-Efficient Gradual Typing in Coercion-Passing Style. In *European Conference on Object-Oriented Programming (ECOOP)*. DOI:http://dx.doi.org/10.4230/LIPIcs.ECOOP.2020.8

Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. 2019. Optimizing and Evaluating Transient Gradual Typing. In *Dynamic Languages Symposium (DLS)*. DOI:http://dx.doi.org/10.1145/3359619.3359742

Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big Types in Little Runtime: Open-World Soundness and Collaborative Blame for Gradual Type Systems. In *Principles of Programming Languages (POPL)*. DOI:http://dx.doi.org/10.1145/3009837.3009849

Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. Integrating Typed and Untyped Code in a Scripting Language. In *Principles of Programming Languages (POPL)*. DOI:http://dx.doi.org/10.1145/1706299.1706343