Sound Gradual Typing: Only Mostly Dead

SPENSER BAUMAN, Indiana University, United States
CARL FRIEDRICH BOLZ-TEREICK, King's College London, United Kingdom
JEREMY SIEK and SAM TOBIN-HOCHSTADT, Indiana University, United States

While gradual typing has proven itself attractive to programmers, many systems have avoided *sound* gradual typing due to the run time overhead of enforcement. In the context of sound gradual typing, both anecdotal and systematic evidence has suggested that run time costs are quite high, and often unacceptable, casting doubt on the viability of soundness as an approach.

We show that these overheads are not fundamental, and that with appropriate improvements, **just-in-time compilers can greatly reduce the overhead of sound gradual typing**. Our study takes benchmarks published in a recent paper on gradual typing performance in Typed Racket (Takikawa et al., POPL 2016) and evaluates them using an experimental tracing JIT compiler for Racket, called Pycket. On typical benchmarks, Pycket is able to eliminate more than 90% of the gradual typing overhead. While our current results are not the final word in optimizing gradual typing, we show that the situation is not dire, and where more work is needed.

Pycket's performance comes from several sources, which we detail and measure individually. First, we apply a sophisticated tracing JIT compiler and optimizer, automatically generated in Pycket using the RPython framework originally created for PyPy. Second, we focus our optimization efforts on the challenges posed by run time checks, implemented in Racket by *chaperones and impersonators*. We introduce representation improvements, including a novel use of *hidden classes* to optimize these data structures, and measure the performance implications of each optimization.

CCS Concepts: • **Theory of computation → Program specifications**; • **Software and its engineering → Just-in-time compilers**; *Software evolution*;

Additional Key Words and Phrases: Gradual Typing, Just-in-Time compilation, Performance Evaluation

ACM Reference Format:
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 (October 2017), 24 pages. https://doi.org/10.1145/3133878

1 REVIVING SOUND GRADUAL TYPING

Gradual typing [Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006] provides a mechanism for migrating dynamically typed programs to statically typed programs incrementally. A gradual transition reduces the upfront cost of transitioning to a statically typed language by allowing static and dynamic code to coexist. These advantages, combined with the popularity of dynamically-typed scripting languages, has led to rapid adoption of gradual typing in industry and in academia, including systems for JavaScript [Bierman et al. 2014; Facebook 2017a; Rastogi et al. 2015; Richards et al. 2015], PHP [Facebook 2017b], Python [Lehtosalo and Greaves 2011; Vitousek et al. 2014],

Authors' addresses: Spenser Bauman, Indiana University, Bloomington, Indiana, United States, sabauma@gmail.com; Carl Friedrich Bolz-Tereick, King's College London, United Kingdom, cfbolz@gmx.de; Jeremy Siek; Sam Tobin-Hochstadt, Indiana University, Bloomington, Indiana, United States, {jsiek,samth}@indiana.edu.

2475-1421/2017/10-ART54
https://doi.org/10.1145/3133878

Dart [Google 2015], Clojure [Bonnaire-Sergeant et al. 2016], Racket [Tobin-Hochstadt and Felleisen 2006, 2008], and others.

Sound gradual type systems guarantee that type invariants (the types specified in statically typed regions of code) are enforced, even in dynamic portions of the program. They therefore require runtime checking of type invariants, with attendant performance costs. Worries about the cost and complexity of enforcement has meant that few gradual type systems are sound, with most ignoring types at run time. Some sound gradual type systems, such as C# [Bierman et al. 2010], StrongScript [Richards et al. 2015], Safe TypeScript [Rastogi et al. 2015], and Lua [Maidl et al. 2014], place significant restrictions on what values can flow between dynamically and statically typed portions of the program to simplify instrumentation and reduce costs..

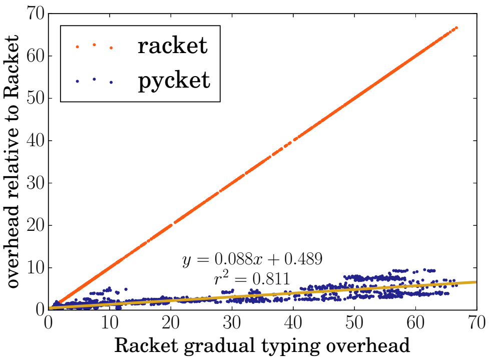


Fig. 1. Performance comparison of Pycket vs the Racket VM across the entire benchmark suite, where each blue dot represents one benchmark configuration.

Are these worries justified? Recently, Takikawa et al. [2015, 2016] set out to answer this question in the context of Typed Racket, the most mature sound gradual type system. Their evaluation method constructs gradually-typed versions of benchmark programs, varying which portions are statically typed. Their results show that the (Typed) Racket implementation measured suffers from substantial slowdowns on many benchmarks, with the vast majority of gradually typed variations suffering a slowdown of more than 3×. In the face of this disappointing result, Takikawa et al. [2016] speculate that sound gradual typing is "dead" if these performance problems are fundamental to the approach.

Happily, we report that gradual typing is not yet "dead", though performance challenges remain. Leveraging Pycket [Bauman et al. 2015; Bolz et al. 2014], a new compiler for Racket that uses tracing JIT compilation, we significantly reduce, and in many cases eliminate, the overhead introduced by gradual typing when compared to the standard Racket implementation, on the benchmarks used by Takikawa et al. [2016].

Figure 1 shows the performance of Pycket and Racket relative to the untyped configuration of Racket (i.e. the configuration without gradual typing overhead). Each dot represents a "configuration", which is one way of assigning types to a benchmark. The x-axis shows the gradual typing overhead experienced by the Racket VM for a configuration, and the y-axis shows the overhead of each configuration relative to the Racket VM's untyped configuration, allowing a comparison of absolute performance. The blue dots represent configurations run under Pycket.

The Racket data follows the line $y = x$, Racket's gradual typing overhead versus itself. Points below the $y = x$ line perform better in Pycket than Racket in terms of absolute runtime. Nearly all configurations are faster in Pycket than in Racket. The Racket VM experiences a maximum overhead of nearly 70×, whereas the (uncommon) worst case for Pycket is just over 10×. The linear regression of the Pycket data in Figure 1 gives an idea of how Pycket's overhead varies relative to Racket:

$$y = 0.088x + 0.489, \ r^2 = 0.811$$

Overall, Pycket suffers a mere 8.8% *the gradual typing overhead of Racket.*

```
(module typed-module typed/racket
  (provide typed-vector)
  (: typed-vector (Vectorof Integer))
  (define typed-vector (vector 1 2 3)))



(module untyped-module racket
  (require typed-module)
  (vector-set! typed-vector 0
               "inconceivable!"))
```

```
(module typed-module racket
  (provide typed-vector)
  (define private-vector (vector 1 2 3))
  (define typed-vector
    (impersonate-vector private-vector
      ;; vector-ref handler
      (lambda (x) x)
      ;; vector-set! handler
      (lambda (x)
        (if (integer? x) x
          (error "expected integer"))))))
```

Fig. 2. Example of typed/untyped interaction resulting from Typed Racket. The left pane shows two modules `typed-module` and `untyped-module`. `typed-module` is written using Typed Racket and defines a vector, `typed-vector`, of type `(Vectorof Integer)`, which is imported into `untyped-module`. The right pane shows the result of Typed Racket's compilation process: a module in plain Racket with `typed-vector` replaced by an vector impersonator.

Pycket owes its performance to multiple factors, which we describe in the remainder of this paper. We begin with an overview of Typed Racket's approach to gradual typing, including Racket's underlying runtime representations for dynamic checks. We then describe the architecture of Pycket, including its JIT compiler and details relevant to gradual typing performance. In Section 4 we review Takikawa et al. [2016]'s benchmarking approach, and show Pycket's overall performance with all optimizations enabled. Section 5 covers specific optimizations relating to gradual typing and evaluates their impact individually. Section 5.2 covers the performance gains to general software contracts as a result of improvements to Pycket. We then describe related work and conclude.

Contributions. This paper contributes to our understanding of gradual typing and compilation in several ways.

- Most significantly, we answer in the affirmative the question of whether sound gradual typing with blame and full interoperability can be made performant. Our system reduces overheads that can be over 100× to typically under 2×. In previous work, whether this was even possible was widely doubted in the literature, and multiple gradual typing proposals have limited expressiveness, error reporting, soundness, or all of the above to avoid this overhead.
- Second, we adapt the technique of *hidden classes* from prototype-based object systems to develop a novel implementation of proxies, a common mechanism used to enforce soundness in gradual type systems. Our approach not only improves the performance of gradually-typed programs, but also other programs that use proxies created by Racket's contract systems [Findler and Felleisen 2002].
- Third, we analyze the sources of Pycket's performance advantages on gradual typing. Our results point the way improving gradual typing performance in other systems as well. Of particular interest is the importance of optimizing loops that traverse proxies.

2 GRADUAL TYPING À LA RACKET

Racket[1] is a dynamically typed language which adds support for static types via Typed Racket [Tobin-Hochstadt and Felleisen 2008], a static type system implemented using Racket's macro system [Tobin-Hochstadt et al. 2011]. At macro expansion time, Typed Racket performs type checking and elaborates the Typed Racket program into untyped Racket code. Typed Racket enables programmers to add type annotations to programs on a per-module basis, so not all modules in

[1]having fallen victim to one of the classic blunders

a program are typed. Figure 2 shows an example program, where a typed and untyped module interact.

The coexistence of typed and untyped code defines gradual typing. Typed Racket provides a *sound* gradual type system, the types specified by Typed Racket are guarantees, and runtime violations of these types in untyped code must be guarded against. The untyped module in the first pane of Figure 2 violates the type assigned to `typed-vector` by assigning the string `"inconceivable!"` to `typed-vector`'s first element. This error cannot be detected statically, as Typed Racket knows nothing about the untyped code in `untyped-module`. Instead, Typed Racket generates code in the form of a contract, which enforces the type annotation in untyped code at runtime.

Typed Racket operates at the module level (often referred to as a *macro-level* gradual typing) and generates the runtime checks required for safe interaction between typed and untyped modules. These runtime checks are expressed using Racket's contract system [Findler and Felleisen 2002]. The creation and enforcement of these contracts is the source of overhead in Racket's implementation of gradual typing.

Data flowing across typing boundaries must be checked to ensure consistency with the specified static types. Some checks occur immediately (e.g. an integer is imported from an untyped module into a typed module). For higher-order values (e.g. procedures, vectors, and structs), the required validation cannot occur immediately.

When a type cannot be checked immediately, values flowing between typed and untyped code are wrapped in an *impersonator*[2] [Strickland et al. 2012], which enforce at runtime the invariants specified using Typed Racket. Impersonators are Racket-level values which proxy other values while presenting the same interface. An impersonator associates handler functions with accessor and mutator operations operating on the underlying values. Handler functions are callbacks which receive the outputs of accessor operations and the inputs of mutation operations. The contract system (and thus Typed Racket) uses the handler functions to validate the inputs and results to procedures and accessor/mutator operations.

Consider again the example in Figure 2. The vector of type `(Vectorof Integer)` flows from `typed-module` to `untyped-module`, so `typed-vector` must be wrapped in an impersonator which protects it from ill-typed operations in `untyped-vector`. A simplified version of the impersonator constructed by Typed Racket for this program is shown in the second pane of Figure 2. The vector impersonator exported by typed-module behaves like the underlying vector, except that the handler functions are invoked when `vector-ref` and `vector-set!` are called on the vector. In Figure 2, the handler provided for `vector-set!` enforces the invariant at runtime by raising an error when the provided value is not an integer.

Impersonators also contain an immutable key/value store which maps *impersonator properties* to arbitrary Racket values. During impersonator construction, an impersonator property may be associated with an arbitrary Racket value; these values may later be retrieved if the key is known. Racket implements impersonator properties using a hash table attached to the impersonator. Impersonator properties are used to recognize, at runtime, where an impersonator was created and provide information on what that impersonator is enforcing. For instance, the contract library (and thus Typed Racket) uses impersonator properties to avoid creating redundant contracts.

Impersonators add overhead in the Racket VM (which is also referred to as Racket) for several reasons:

- extra pointer indirections to access the underlying data,
- extra runtime checks to enforce the static types at runtime,

[2]Racket distinguishes between two types of proxies, chaperones and impersonators, which are collectively referred to in this paper as impersonators. The distinction between the two proxy types is not relevant to this paper.

- Racket's JIT cannot produce code specialized to non-trivial impersonator structures, and
- impersonators and associated data structures constitute a significant increase in overall memory allocation.

These problems are further compounded when data flows across multiple typed/untyped boundaries, accumulating multiple (possibly redundant) layers of impersonators. Work is further duplicated as Typed Racket elaborates to normal, dynamically typed Racket, where the runtime system will perform many of the same checks introduced by Typed Racket.

2.1 Racket's Optimizations for Gradual Typing

Improvements to (Typed) Racket have substantially reduced the overhead of gradual typing since Takikawa et al. [2016]—improvements which also benefit Pycket. These improvements come in several varieties, all of which have significant impact on at least one benchmark. The data we present in this paper all includes these optimizations.

Fewer wrappers. Typed Racket avoids generating impersonator wrappers for functions when they are used in a first-order way, relying on optimizations in the underlying contract system. Avoiding such wrappers reduces both allocation and indirection overhead. Since these wrappers are immediately consumed by the function call, this optimization is less needed in Pycket than in Racket.

Specialized contracts. Takikawa et al. [2016] note that many of the generated contracts check a few simple types, notably `(Any -> Boolean)`. In many cases, these contracts were applied to known procedures such as structure predicates which necessarily satisfy these contracts, allowing them to be safely omitted.

Runtime support. The Racket virtual machine provides an unchecked constructor for procedure wrappers which avoids some indirections and dynamic checks, as well as enabling further compiler optimization.

Representation improvements. The Racket contract system avoids conversion between different representations of contract values, which is significant for higher-order contracts such as vectors, particularly on the synth benchmark.

These optimizations have a major effect on Racket's performance, and were designed with that end in mind, but also affect the performance of Pycket.

3 PYCKET BACKGROUND

Pycket [Bauman et al. 2015; Bolz et al. 2014] is an experimental Racket implementation developed using the RPython tool chain [Bolz et al. 2009; Rigo and Pedroni 2006]. The design of Pycket is unusual compared to most performance oriented language VMs. Rather than implement a bytecode machine, Pycket directly interprets Racket's AST after macro expansion with a CEK machine [Felleisen and Friedman 1987].

Pycket operates on Racket's core forms, which are the end result of the macro expansion process [Tobin-Hochstadt et al. 2011]. Rather than re-implement Racket's macro expander, Pycket makes use of the Racket VM's existing macro expander. For the purposes of gradual typing, this means Pycket's performance is partially dependent upon the code generated by Racket and its associated libraries. Thus, the choice of Racket version (particularly the associated version of Typed Racket) affects Pycket's performance. Unless otherwise stated, Pycket results make use of Racket 6.6, which includes all of the changes mentioned to (Typed) Racket.

Pycket performs little manipulation to the actual input program before execution. The only alterations performed to the AST before interpretation are assignment conversion [Adams et al. 1986], and conversion to A-normal form (ANF) [Danvy 1991; Flanagan et al. 1993]. Pycket converts to ANF to simplify the interpreter implementation, while assignment conversion places all mutable variables into heap allocated cells, which allows for an immutable environment representation.

Pycket's performance depends heavily on the RPython meta-tracing JIT compiler [Bolz et al. 2009], a tracing JIT generated from an interpreter implementation. Trace compilation generates machine code for the control-flow paths recorded during execution, implicitly performing type specialization and inlining. This aggressive inlining and specialization makes tracing well suited to optimizing gradually typed programs. The RPython JIT is able to propagate the information gained from executing runtime checks (facilitated by impersonators) to elide subsequent type tests which are part of Racket's normal execution. As long as the handlers performing the additional runtime checks occur on the same trace as the code consuming the results, these additional runtime checks pay for themselves.

3.1 Loop Detection

Loop detection is an important aspect of efficient code generation in a tracing JIT compiler. In many imperative languages, loop detection is simplified to a purely syntactic property of the program, as loops can only be formed by a small number of looping constructs (`for`, `while`, `do`, etc.). In Racket, loops are constructed through functions and recursion. Rather than attempt to detect loops in the program using an imprecise static analysis, Pycket detects cycles dynamically.

Pycket has two loop detection mechanisms [Bauman et al. 2015], the call-graph and two-state tracking. Two-state tracking filters false loops by associating traces to caller-callee pairs. In the absence of the call-graph, all functions are annotated as potential loops for the JIT. False loops are points a program's control flow which are erroneously identified as loops by the JIT.

The call-graph builds a graph representing the caller-callee relationship of all functions in the program invoked during execution. During execution, Pycket records all function calls as caller-callee pairs as edges in a directed graph. When adding a new edge introduces a cycle, the target of the new edge (analogous to a back edge in the control flow graph) is marked as a loop header for the JIT.

The call-graph, however, is essential for obtaining acceptable performance for Racket's contract system. The heavy use of higher-order functions to validate data obscures source level loops in the program, requiring multiple traces (optimized independently) to cover the whole loop. The call-graph is able to properly detect loops in the presence of impersonators and significantly improve performance over two-state tracking used in isolation (see Section 5.3).

3.2 Proxies in Pycket

Pycket's implementation of impersonators closely follows their high level specification. Operating on an impersonator involves looking up the appropriate handler, and pushing a continuation frame to receive the result of the handler.

Each impersonator type is designed to wrap one of Racket's built-in value types (e.g. vectors, procedures, hash maps, structs, etc.). An impersonator associates handler functions to a type's accessors and mutators. For example, a vector impersonator associates a handler to the `vector-ref` which is invoked on the result of the `vector-ref`, and a handler to the `vector-set!` operation which is invoked on the inputs of `vector-set!`. As a more complex example, hash table impersonators can have up to six handlers associated to the operations: `hash-ref`, `hash-set!`, `hash-remove!`, `hash-iterate-key`, `hash-clear!` and `equal?`.

In Pycket, all associated handlers are stored inline on the impersonator, except for struct impersonators. Structs can have an arbitrary number of fields, and each field can have an associated accessor and mutator. In addition, structs have a key/value store of their own associated with each struct type, called *struct properties*, whose accessor operation can have an associated handler. Struct impersonators are more complicated than impersonators for other built-in data types.

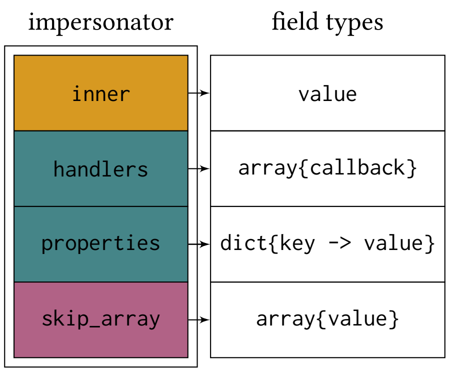


Fig. 3. Baseline layout of struct impersonators

All impersonator types except those for user-defined structs store a small, fixed number of handlers and their impersonator properties. For most impersonator types, Pycket stores handler functions inline on the impersonator, and impersonator properties are stored in a hash table, much like Racket's internal representation.

Struct impersonators are more complicated in Pycket than other impersonator types due to the complexities of Racket's struct system. Figure 3 shows the layout a struct impersonator and the corresponding field types. The `inner` field points to the underlying struct (which may also be a struct impersonator). The `handlers` array contains the callback functions for each field, while the `properties` dictionary stores impersonator and struct property handlers. Finally, the `skip_array` field is used to handle long impersonator chains.

Though struct impersonators are frequently sparse – only a subset of their accessors and mutators have associated handlers – lookup of the associated handler must be fast. For that reason, baseline Pycket allocates an array with sufficient storage for all accessors and mutators, making handler lookup an array access. The downside is that Pycket allocates more storage than is needed in order to provide fast lookup.

Impersonator and struct property handlers are stored in hash tables on each impersonator, incurring the cost of allocation and hash table lookup to find a handler.

Long chains of impersonators are more common for structs than for other Racket data types. One benchmark from Strickland et al. [2012] produces impersonator chains of length 2000. Traversing each level of such long impersonator chains is expensive and can result in an exponential growth in code size if traced by the JIT. To avoid this problem, Pycket leverages the sparsity of most struct impersonators by constructing a jump table (the `skip_array` field in Figure 3) at each level which points to the next impersonator implementing each handler. This allows Pycket to skip over large portions of such impersonator chains at the cost of additional allocation.

Sometimes operating on deeply nested impersonators cannot be avoided. In these cases, Pycket must take care when to not inline the long impersonator traversals into the trace. A field read on a long impersonator chain has the effect of unrolling the reference operation as it traverses the entire chain. To avoid exponential code growth, in the event that all impersonators in the chain must be visited, Pycket uses the RPython JIT to convert impersonator traversal into a separate loop when operating on deep proxies.

4 BENCHMARKS

This section presents an overall evaluation of Pycket's performance, with all optimizations enabled. The effects of specific optimizations are evaluated in Section 5.

To validate our claims about Pycket, we make use of the evaluation method developed by Takikawa et al. [2016] as well as their suite of gradual typing benchmarks. This benchmark suite

	Racket			Pycket			
benchmark	max overhead	mean overhead	% configs < 2×	max overhead	mean overhead	% configs < 2×	Pycket/ Racket
`sieve`	30.1	9.1	50	4.2	2.4	50	0.75
`morse-code`	1.9	1.3	100	1.1	1.0	100	0.80
`mbta`	2.3	1.6	75	1.5	1.3	100	0.22
`zordoz`	3.4	1.6	75	1.5	1.1	100	0.84
`suffixtree`	19.3	10.3	16	9.6	3.8	44	0.45
`kcfa`	6.3	3.4	25	3.6	2.1	49	0.37
`snake`	12.7	6.8	5	5.7	1.9	74	0.91
`tetris`	11.8	5.4	25	2.5	1.5	86	0.43
`synth`	66.7	36.7	1	10.5	4.1	16	0.90
`gregor`	2.1	1.6	100	1.7	0.5	100	1.18

Fig. 4. Summary statistics corresponding to the L=0 slowdown lines (solid) Figure 5 for Racket and Pycket on code generated by Racket 6.6. These statistics include max and mean gradual typing overhead, and the percentage of configurations at < 2× slowdown. The final column lists the runtime of Pycket divided by the runtime of Racket VM on the untyped configuration.

consists of twelve programs of varying size and complexity. Currently Pycket supports ten of the twelve: `sieve`, `morse-code`, `mbta`, `zordoz`, `suffixtree`, `kcfa`, `snake`, `tetris`, `syth`, and `gregor`. This includes *all* programs on which Takikawa et al. [2016] observe a gradual typing overhead of greater than 3×. The remaining two programs (`lnm` and `quad`) are not supported due to their reliance on Racket's FFI and GUI libraries, which Pycket does not currently support. However, these benchmarks do not suffer from substantial gradual typing overhead. The next section provides a recap of the benchmark design and evaluation method developed by Takikawa et al. [2016].

4.1 Background

To evaluate the effects of gradual typing on Pycket's performance, we make use of the methodology developed by Takikawa et al. [2016]. This process involves examining the performance of all possible typing configurations for each benchmark. A benchmark is a collection of modules, and a configuration is one possible way to assign types to that benchmark. Each benchmark contains two versions of every module: typed and untyped. The typed and untyped modules are then combined in all possible ways to generate all possible ways of assigning types to the benchmark.

Each configuration is uniquely identified by a bit-string indicating whether modules are untyped or typed. A benchmark (such as `sieve`) with 2 modules has 4 total configurations: `00`, `01`, `10`, and `11`. Configurations `00` and `11` are the untyped and fully typed configurations, respectively, while `01` and `10` are the gradually typed configurations. More generally, a benchmark with n modules has 2^n configurations.

The configurations of a benchmark can be arranged into a lattice with the untyped and fully typed configurations as bottom and top elements, respectively. The i^{th} level of the lattice contains all configurations with i typed modules. A configuration at level i is related to another at level $i + 1$ if that configuration may be obtained by typing one additional module. Each path through the lattice represents one way to transition the benchmark from untyped to fully typed.

The question we wish to answer then is, "does Pycket significantly increase the total number of acceptable configurations?" What constitutes an "acceptable" configuration is application specific, as different systems have distinct performance requirements. The *slowdown* of a configuration due to gradual typing is the runtime of that configuration divided by the runtime of the untyped configuration of the same benchmark. Takikawa et al. [2016] chose the generous value of 3× slowdown as the definition of acceptable.

Configurations which perform poorly may be fixed by typing a small number of additional modules. To address this, Takikawa et al. [2016] define the *L-step* performance of a configuration to be the best performance achievable by typing at most L additional modules. Takikawa et al. [2016] examined the performance of Racket out to L=2; we focus only on L=0 and L=1, since Pycket's performance improvements reduce the need to type multiple modules to achieve good performance.

Diverging from Takikawa et al. [2016], this paper will avoid using a 3× overhead as the definition of acceptable; what constitutes "acceptable" is too application specific to generalize in a meaningful fashion. Such a definition is not necessary, as the CDF based slowdown plots developed by Takikawa et al. [2016] present the data in a form which allows for an arbitrary choice of acceptable threshold. Compared to the Racket VM, Pycket renders more configurations performant, reducing the likelihood of encountering an unacceptable configuration while transitioning from untyped to fully typed. In the event that an unacceptable configuration is encountered, Pycket reduces the work required to further transition to an acceptable configuration. Invariant of the definition of acceptable, Pycket represents a significant improvement in performance over existing implementations of gradual typing.

Experimental Setup. Due to the large number of configurations of some of the benchmarks, all benchmarking was performed on the Karst computing cluster at Indiana University. Each of the 228 general purpose compute nodes is equipped with 2 Intel Xeon E5-2650 v2 processors with a total of 16 CPU cores and 32 GB of RAM, running Red Hat Linux. Each benchmark configuration was run 16 times and averaged to get the final runtime. Benchmarks were scheduled to ensure that only one benchmarking instance was active on a node to prevent resource contention.

Accounting for warmup in a complex VM which performs speculative optimizations at runtime is a not straightforward [Barrett et al. 2016]. Most of the warmup time experienced by Pycket is not fundamental to trace compilation. High warmup time is a consequence of Pycket's high level architecture and must be accounted for in order to demonstrate Pycket's peak performance.

To account for warmup time in Pycket, benchmarks ran for five iterations (`gregor`, `mbta`, and `zordoz` ran for ten), taking the runtime of the last.[3] The commit SHA for Pycket used to generate these results is `3573fd1`. All versions of Pycket were built using RPython revision `7c64684c80f4`. Racket 6.6 makes use of Typed Racket using revision `ec0c851`. Some figures include results using code generated by Racket 6.2.1, the version of Racket used by Takikawa et al. [2016].

4.2 Gradual Typing Overhead

Figure 5 illustrates the slowdown (relative to the untyped configuration) introduced by gradual typing, similar to Figure 4 from Takikawa et al. [2016] The x-axis gives the slowdown factor relative to the untyped configuration, while the y-axis gives the number of configurations at or below that slowdown factor. A benchmark with zero gradual typing overhead would consist of a vertical line at $x = 1$ which then extends horizontally along the top of the plot. Each plot in Figure 5 shows the results for L=0 and L=1 for Pycket and Racket. Note that all the benchmarks shown in Figure 5 do not include warmup time for Pycket. Due to the architecture of Pycket, warmup time contributes

[3]The `gregor` benchmark (see Section 4.3) has an exceptionally high warmup overhead (approximately 12× on the untyped configuration). This is due to an unfortunate interaction between trace compilation and Racket's numeric tower; it is not relevant to gradual typing performance but must be accounted for in the benchmarks.

significantly to overall runtime of some benchmarks, which obscures the overhead due to gradual typing.

Across all benchmarks Pycket averages 2.0× slowdown, while Racket averages 7.8×. Racket has a large improvement from L=0 to L=1, but it is insufficient to achieve performance equivalent to Pycket. A full 72% of all configurations experience less than a 2× slowdown in Pycket and that number increases to 94% if one additional module is typed, whereas just 47% are below 2× slowdown in the Racket VM at L=1 and 62% at L=2. If we consider configurations experiencing less than a 10% slowdown, 45% of all configurations are usable in Pycket, while 24% are usable using the Racket VM (increasing to 66% and 76%, respectively, if one additional module may be typed). Even with these more stringent requirements, a sufficiently performant configuration may be reached by typing up to two additional modules 77% of the time.

Finally, Figure 4 contains a table of summary statistics for the data used to generate Figure 5. The Figure includes the max and mean gradual typing overhead, the percentage of configurations experiencing less than a 2× slowdown, and the final column shows the ratio of Pycket's runtime over Racket's runtime on the untyped configuration for each benchmark.

4.3 Individual Benchmarks

Sieve. is a implementation of the Sieve of Eratosthenes using lazy streams. This is the smallest benchmark, consisting of two modules (four configurations, two gradually typed). Neither system produces good performance on the gradually typed configurations, though Pycket improves their performance, decreasing maximum overhead from 30.1× to 4.2×.

Morse code. is a Morse code training program. The benchmark generates Morse code strings and runs the Levenshtein algorithm on a list of frequently used words. Racket keeps all configurations under 2× slowdown, while Pycket keeps all configurations 1.1× slowdown.

MBTA. performs reachability queries on Boston's public transit system, using an untyped graph library. Pycket reduces the maximum overhead from 2.3× to 1.5×. In addition, Pycket is over 4× faster than Racket on the untyped configuration.

Zordoz. is a tool to explore the structure of the Racket VM's bytecode representation. Most configurations experience little to no slowdown in both Pycket and Racket. On the small number of configurations which experience more than 3× slowdown, Pycket limits the overhead to 1.5×.

Suffixtree. measures the performance of Ukkonen's suffix tree algorithm used to compute the longest common substring. The performance of `suffixtree` is negatively impacted by an optimization in Typed Racket, resulting in the bimodal slowdown distribution in Figure 5. `Suffixtree` creates many immutable vectors which must be checked at typing boundaries. The contract system avoids creating vector impersonators for immutable vectors by checking the vector contents eagerly. The validation loop presents an optimization boundary for a tracing JIT, which operates on a per-loop basis. Rather than trace through an impersonator allocation, the JIT must stop tracing and jump to the validation loop. In Pycket, lazy checking of contracts is faster, as fewer loops are introduced and the checks can be fused with code operating on the values being checked. When said optimization is disabled, mean overhead drops to 1.9×, with a maximum overhead of 3.6×.[4]

KCFA. performs control flow analysis for a λ-calculus. Pycket reduces the max overhead from 6.3× to 3.6× and is nearly 2.7× faster on the untyped configuration.

[4]For fairness of comparison, the optimization was not removed for Pycket, as doing so has a deleterious effect on the Racket VM's performance.

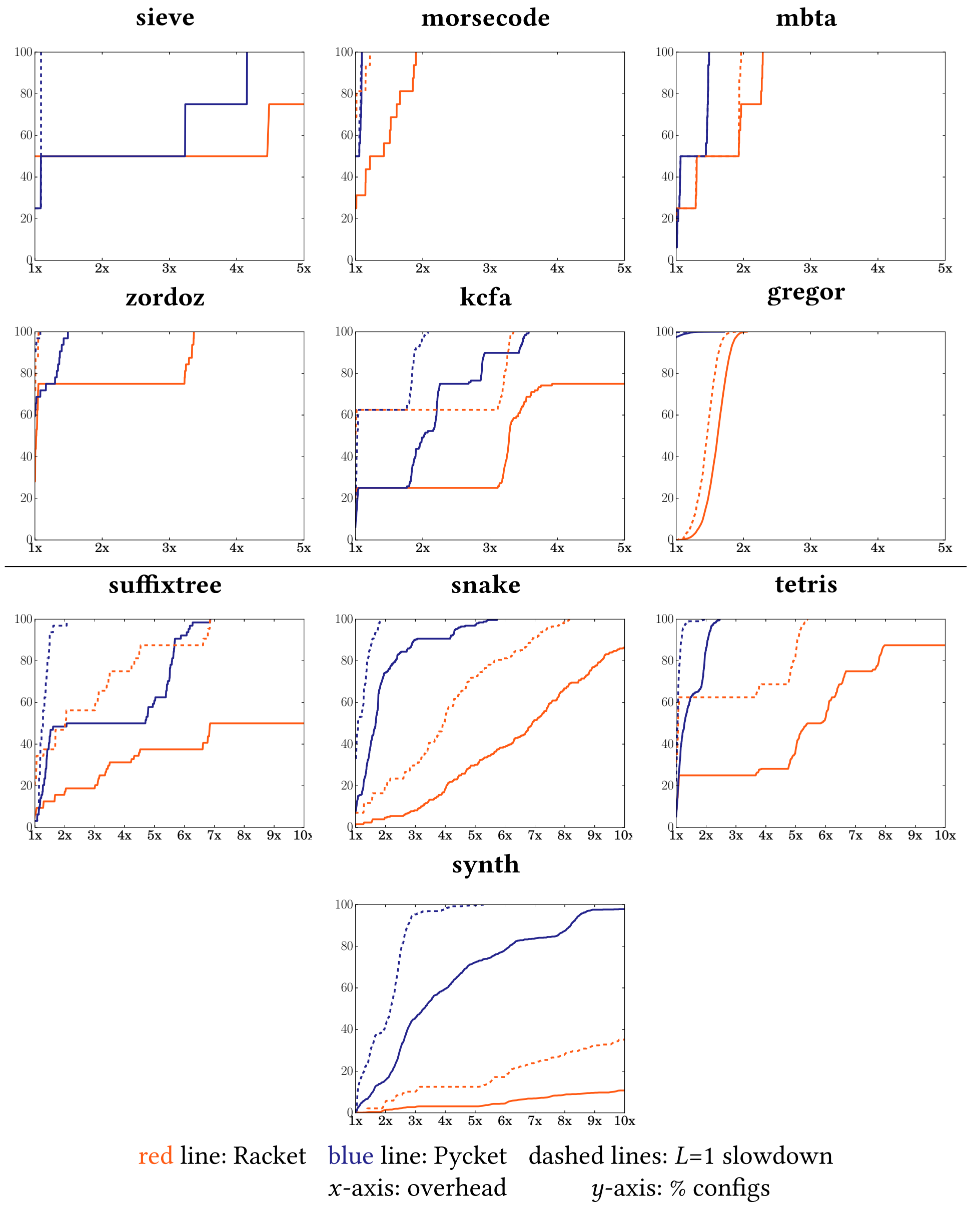


Fig. 5. L-step slowdown results

Snake. simulates the well known game involving a snake which grows by consuming apples. The benchmark uses a pre-recorded sequence of moves. The Racket VM exhibits a mean slowdown of 6.8× and a maximum slowdown of 12.7×. Most configurations (over 70%) experience less than a 2× slowdown in Pycket, with a mean overhead of just 1.9×. Pycket has a small number (about 10%) of outlier configurations which extend out to a 5.7× slowdown.

Tetris. is similar to snake, simulating the well known game of the same name. Pycket reduces the maximum overhead from 11.8× down to just 2.5× with a mean overhead of just 1.5×.

Synth. performs sound synthesis, making heavy use of array and numeric operations. This benchmark is the most heavily affected by gradual typing for both Pycket and Racket. A mere 1% of all configurations have below a 2× slowdown in the Racket VM, with a maximum overhead of 66.7×! Pycket fares better, with a maximum overhead of 10.5×, but only 16% of all configurations experience less than 2× slowdown. While this represents a significant improvement over the Racket VM, there is still considerable room for improvement.

Gregor. This benchmark is the largest which Pycket supports, consisting of 13 modules which implement and test a date/time library. As mentioned previously, `gregor` has significant warmup overhead not related to gradual typing. The Racket VM experiences a mean overhead of 1.6× on this benchmark, while Pycket experiences a mean overhead of just 0.5×.

Pycket is strictly an improvement over Racket when considering gradual typing overhead for every metric. For a given slowdown, Pycket renders more configurations usable. The mean gradual typing overhead is likewise reduced (by nearly 9× on `synth`), and the maximum overhead is similarly improved. With the exception of `synth`, all configurations can be kept to just over a 2× slowdown by typing one additional module.

4.4 Absolute Performance

The results of the previous section show that Pycket's performance degrades less than Racket's in the presence of gradual types. One possible explanation is that Pycket is uniformly slow on all benchmarks, rendering the results of the previous section singularly unimpressive. This notion is readily dispelled by Figure 1, which compares Pycket's performance relative to the Racket VM on a per configuration basis. All data points below the line formed by the Racket data are faster in terms of absolute runtime. A small number of configurations which have relatively low overhead in Racket perform better than Pycket, while *configurations with more than* 2× *overhead in Racket are always faster in Pycket.*

	# of impersonators		runtime (ms)	
configuration	procedure	struct	hidden	baseline
`00`	0	0	9162	9184
`01`	100856180	20	38644	52753
`10`	64	20	28949	29569
`11`	50	16	9924	10157

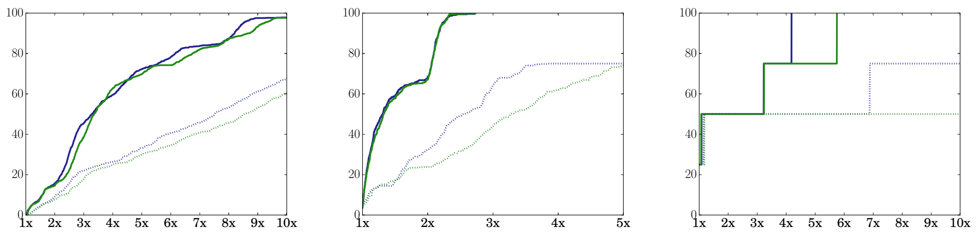
Fig. 6. Number and type of impersonators generated by all four configurations of the `sieve` benchmark. The final two columns show Pycket's runtime with (hidden) and without (baseline) hidden classes in the implementation of impersonators.

The final column of Figure 4 shows Pycket's runtime on the untyped configuration divided by Racket's runtime on said configuration, for each benchmark. On only one benchmark, `gregor`, is Pycket slower than the Racket VM for the untyped configuration. This is due to the nature of trace compilation; the order in which hot code paths are detected and optimized can have a significant effect on performance. The `gregor` benchmark exacerbates this problem by having a very large number of code paths. Poorly optimizing the untyped configuration is what results in the 0.5× mean slowdown under Pycket. Note that the 20% slowdown relative to Racket on the untyped configuration is insufficient to account for the 3× improvement in mean gradual typing overhead.

On the other hand, the remaining eight benchmarks are all faster in Pycket (often substantially) than in the Racket VM. Yet in all cases, Pycket suffers less gradual typing overhead. This shows that Pycket suffers less overhead due to gradual typing and is also faster in terms of absolute runtime.

5 OPTIMIZING GRADUAL TYPING

This section outlines the features of Pycket which improve the performance of gradual typing. The results demonstrate how optimizations in Pycket affect the performance of gradually programs.



(a) synth (b) tetris (c) sieve

Fig. 7. Gradual typing overhead on Pycket with and without hidden classes on `synth`, `tetris`, and `sieve`. The blue and green lines are Pycket with and without hidden classes, respectively. The dotted lines show results for code generated by Racket 6.2.1., in which more impersonators are generated.

5.1 Hidden Classes

Optimizing the representation and use of impersonators is crucial to gradual typing performance. The naïve implementation of impersonators has several sources of overhead.

- **indirection:** Accessing the handler procedures for struct impersonators must go through an additional array or hash table lookup.
- **memory allocation:** All impersonators with at least one impersonator property allocate a hash table. Struct impersonators allocate either arrays or hash tables to store handlers.
- **hash table lookup:** Impersonator and struct property access performs a hash table lookup, even though the key set for these properties is statically determined in most cases.

To solve these problems, we developed a new implementation of impersonators which allows for a more compact memory layout and enables the generation of efficient machine code by the JIT compiler.

The new representation uses a hidden class [Bolz et al. 2011; Chambers et al. 1989] to describe the layout of each impersonator while storing all data in a contiguous storage area. In an object oriented languages like JavaScript and Python, hidden classes map field names to integer values, which are then taken as offsets into a contiguous storage location. Abstractly, a hidden class is a datum which describes an object's layout by mapping attributes to storage locations with the condition that all objects with the same layout have the same hidden class. Figure 9 shows the data layout of the new, hidden class based, struct impersonators.

For impersonator types with a fixed number of handlers (all except struct impersonators), Pycket stores the handler routines inline as before, but now impersonator properties are also stored inline on the impersonator, at locations described by the hidden class. The immutability of impersonators allows Pycket to perform the same size and type specialization used in other key runtime data structures [Bauman et al. 2015].

Pycket's use of hidden classes differs slightly from standard presentations, due to the different types of data which can be stored in struct impersonators. Three cases must be handled by Pycket's implementation of hidden classes: (1) handler functions for struct fields, (2) handler functions for struct properties, (3) keys for the impersonator property store. The hidden class maps field indexes (integer values) to the corresponding storage locations. The field index is first tagged to encode whether a given handler is for an access or mutation operation (simply `index = 2` $\times$ `field + is_accessor`).[5]

[5] The actual implementation is slightly more complicated, as a struct field may have up to four associated handler functions.

Benchmark	Racket	Baseline	Hidden Classes	Speedup vs Baseline	Speedup vs Racket
Bubble					
direct	1016	587	588	1.00	1.73
impersonator	5120	613	628	0.98	8.15
Church					
direct	1342	581	587	0.99	2.29
impersonator	31816	4645	2087	2.23	15.2
contract	5228	879	809	1.09	6.46
wrap	3140	2755	2763	1.00	1.14
Struct					
direct	266	110	110	1.00	2.42
impersonator	1012	106	108	0.98	9.37
unsafe	260	105	105	1.00	2.48
unsafe*	256	106	105	1.01	2.44
ODE					
direct	9964	1348	1351	1.00	7.38
contract	12054	1561	1533	1.02	7.86
Binomial					
direct	2348	1225	1092	1.12	2.15
contract	41904	10899	8202	1.33	5.11

Fig. 8. Runtime (in ms) of Racket and Pycket, with and without the hidden class based representation of impersonators. The final two columns show the speedup of hidden classes over Racket and the baseline impersonator implementation.

The hidden class must also map struct properties and struct types to the corresponding storage locations for their handler operations. Struct properties are immutable, so there is only an access handler. Finally, the hidden class must also map impersonator properties to the storage locations for the corresponding values.

Unlike objects in most dynamic languages, Racket's impersonators are immutable. This allows the implementation to allocate the minimum amount of space necessary, without worrying about fields being added or removed at runtime which would make it necessary to mutate the hidden class field.

This new representation saves the allocation of three arrays and up to two hash tables. Using hidden classes, the JIT generates fast (a single pointer offset) handler retrieval code while storing all handlers contiguously. The common case is that all struct and impersonator property lookups have static keys, which the JIT converts into an identity comparison on the hidden class followed by a single field read on the impersonator.

Hidden classes also replace the array used to skip portions of the impersonator chain which do not have handlers associated with an accessor/mutator. Rather than store an array of pointers for each accessor/mutator, the hidden class implementation of struct impersonators stores a pointer to the next impersonator in the chain with a *different* hidden class (the `skip` field in Figure 3). Because the common case is that all impersonators in a chain have (or lack) handlers for the same accessors and mutators, this strategy is sufficient to elide traversal of large portions of long impersonator chains.

The performance impact of hidden classes varies significantly between benchmarks and versions of Typed Racket. Many of the benchmarks shown in Figure 5 are unaffected by hidden classes due to improved code generation in recent versions of Typed Racket. *Across all benchmarks, the mean overhead increases from* 1.97× *to* 2.05× *when hidden classes are disabled.*

Figure 6 gives runtime statistics collected from the `sieve` benchmark. `Sieve` is small enough that we can provide information on the execution of all configurations. Configurations `00` and `11` are the untyped and fully typed configurations, respectively.

This leaves `01` and `10` as the two gradually typed configurations. Configuration `01`, which produces over 100 million[6] impersonators whilst calculating the 10000th prime number, demonstrates the performance improvements due to hidden classes. Hidden classes reduce Pycket's overall runtime by 30%. These savings come from the combination of reduced allocation and exchanging hash table operations for pointer comparison (as described in Section 5.5). Configuration `10` produces far fewer impersonators, resulting in little benefit due to the use of hidden classes.

CDF plots for the benchmarks most affected by the use of hidden classes are shown in Figure 7. The `tetris` panel shows how improvements to Typed Racket have reduced the need for hidden classes in some programs. Without the overhead of heavy impersonator use, the performance benefits are not observable. The slowdown distribution for both Pycket variants are indistinguishable using code generated by Racket 6.6, but using code from Racket 6.2.1, the new impersonator implementation is a notable performance improvement for all three benchmarks. Though some programs do not benefit from hidden classes, hidden classes have little overhead and never have a negative performance impact once warmup time is removed.

5.2 General Contract Performance

As noted previously, Pycket operates below the level of Typed Racket language and gradual types, and does not optimize them directly. Rather, Pycket implements and optimizes impersonators, which are generated by the contract system. For the benchmarks shown in Figure 5 the improved impersonator representation has little impact, as Racket's gradual type system does a decent job of allocating as few impersonators as possible. Such optimizations are difficult to do for contracts in general, which can enforce arbitrarily complex properties.

In the general case, the performance implications of the improved impersonator implementation are more significant than the benchmarks above suggest. For example, Racket's contract benchmark suite [Strickland et al. 2012] benefits from the new impersonator representation.

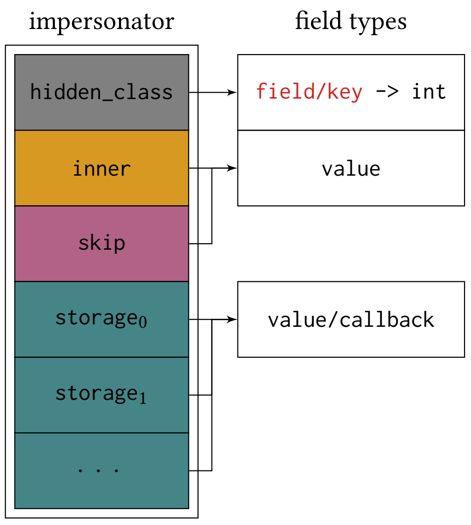


Fig. 9. Hidden class based layout of struct impersonators

Figure 8 show the performance of the Racket VM and Pycket, with and without hidden classes, on the contract benchmark suite. The variants of each benchmark labeled "impersonator"

[6]`Sieve` is not a faithful implementation of the Sieve of Eratosthenes, having suboptimal asymptotic behaviour. This causes the large allocation of closures and corresponding impersonators.

or "contract" both make use of impersonators; "impersonator" uses hand written code for generating impersonators, while "contract" uses impersonators generated by the contract system. Many variants experience no performance difference, even when using impersonators. In those cases, the JIT compiler is able to elide most of the work associated with impersonators, regardless of the chosen representation.

The highlighted rows of Figure 8 show performance improvements due to the use of hidden classes. These benchmarks are all variants of the **Church** and **Binomial** benchmarks. **Church** is a benchmark which computes the factorial function using Church numerals. **Binomial** benchmarks the performance of a binomial heap in the context of an image processing application. In both cases, a large number of impersonators must be allocated, which is the case where hidden classes have observable performance benefits.

At a high level, the impersonator and contract variants of **Church** enforce the same property. The performance discrepancy between the two versions is due to optimizations in the contract system which avoid producing redundant wrappers. The impersonated variant of **Church** improves by a factor of 2.23× thanks to the hidden class based representation, while the variant using the contract system improves by only a factor of 1.09×.

The contracted version of **Binomial** improves by a factor of 1.33×. While this is a substantial improvement, there is still significant overhead when compared to the direct implementation. This is due to the large number of impersonators generated for each node in the binomial heap data structure. The contracts in this benchmark ensure that the binomial heap stays balanced. By the end of the program, some nodes in the heap have over *2000* layers of impersonators. While the hidden class implementation reduces indirection and allocation, it cannot eliminate the overhead of such an inefficient use of impersonators.

While hidden classes improve performance of some contracts significantly when compared to the baseline implementation, both impersonator implementations are substantially faster than the Racket VM. Pycket is faster than the Racket VM on all variants of each benchmark, even those without impersonators. When impersonators are used in a benchmark, either manually or through the contract system, the performance gap between Pycket and the Racket VM increases.

5.3 The Call-Graph

The call-graph mechanism for loop detection [Bauman et al. 2015] is essential for obtaining good performance on gradually typed programs. The use of impersonators to ensure correctness makes loop detection difficult. The impersonators which ensure soundness are generated by Typed Racket using Racket's contract library.

While the call-graph constitutes prior work, it is important to note the significance of the call-graph in relation to gradual typing. The performance impact of the call-graph, versus less complex approaches detailed by Bauman et al. [2015] is minimal for most benchmarks presented therein. Dynamic loop detection is uncommon, even for tracing JITs, so it is worth noting how crucial the call-graph is to Pycket's success on gradually typed programs.

Without the call-graph, tracing frequently begins in the runtime functions used by the contract system. These functions are seen as "hot" by the JIT, as they are executed frequently, as gradually typed programs make extensive use of the contract system. These runtime functions are code paths which are shared by many contracts generated by the contract system and Typed Racket. Because the code for the contract system is shared so heavily, contract checking performs best when specialized on the usage site in the running program.

To specialize the code of the contract system on its usage site, tracing must begin in the user's program (not in the code for the contract system) and trace through contract enforcement. Tracing through contract enforcement inlines and specializes the relevant code paths needed to enforce a particular contract on its calling context.

The call-graph prevents this problem by identifying loops in the users program and marking only the loops for compilation. This prevents tracing for starting in the functions involved in contract enforcement. Tracing from the small, heavily used functions in the contract library creates an optimization barrier by segmenting actual loops in the user's program into multiple traces, which must be optimized independently. This problem is similar to the regression of the `suffixtree` benchmark in Pycket due to a Typed Racket optimization discussed in Section 4.3.

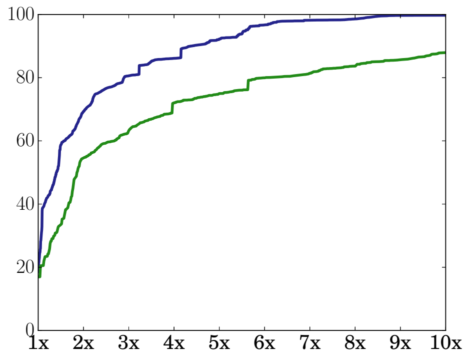


Fig. 10. Slowdown CDF across the entire benchmark suite. Each benchmark is allotted 10% of the y-axis. The blue line is the standard Pycket implementation while green line is Pycket without the call-graph.

In contrast to the Racket VM, Pycket performs better when checking occurs lazily for large data structures like immutable `vectors`.[7] Lazy checking removes the data-validation loop from module boundaries by using an impersonator to perform the check at the point where the data is accessed. This has two performance implications in a tracing JIT:

- the potentially expensive validation loop omitted entirely,
- the addition runtime type checks are inlined and optimized out by the JIT.

The effect is similar to using trace compilation to perform deforestation and loop-fusion in lazy functional languages [Schilling 2013].

Figure 10 shows the effect of the call-graph across the entire benchmark suite. This combined figure weights each benchmark so it occupies the same space on the y-axis, regardless of the total number of configurations.[8]

For the entire benchmark suite, disabling the call-graph increases the mean gradual typing overhead from 2.08× to 3.70×. Note that the slowdowns are computed relative to the untyped configuration *for each system*. In a few cases, disabling the call-graph slows down the untyped configuration as well, reducing the mean slowdown. If instead we compute the slowdown relative to the standard Pycket implementation, then the mean slowdown increases to 4.6×.

5.4 JIT Improvements

Pycket stresses the RPython JIT quite differently from PyPy, its original target. Traces are much longer, relying on the optimizer to eliminate redundant operations. Impersonators further stress the optimizer, resulting in failed optimizations and suboptimal code. A particularly brittle optimization which Pycket depends on is the loop-peeling transformation which allows the JIT to perform loop invariant code motion [Ardö et al. 2012].

[7]In the absence of type errors, this strategy is equivalent to eager checking. With lazy checking, fields of inputs may be ill-typed if they are never accessed.

[8]Data for `gregor` is not included as it is not available for the Pycket without the call-graph due to the large runtime of `gregor`.

This transformation works by appending a trace to itself and optimizing the result as a single compilation unit. Two traces are produced this way, the *preamble* containing all loop-invariant operations, and the *peeled-iteration* containing only the non-loop-invariant operations. In Pycket, the majority of loop-invariant operations come from the interpreter state, code in the preamble must deconstruct the interpreter state.

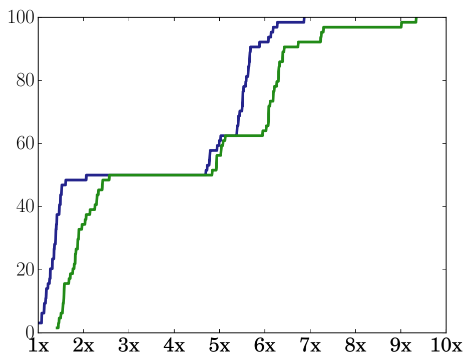


Fig. 11. Comparison of gradual typing overhead on the `suffixtree` benchmark for Pycket with and without improvements to the loop-peeling transformation. The blue and green lines show Pycket with and without the optimization, respectively.

As far as optimizing loops goes, this optimization works well enough. However, problems occur as the control flow becomes more complex. If a side exit of a loop is traced, this side trace often jumps back to the original loop. By default, the side trace would jump to the preamble, which means that if control follows this path, the loop-invariant operations are performed again. Instead, the JIT performs an analysis to determine whether a trace may jump to the peeled-iteration.

When jumping to the preamble, it is first necessary to allocate any unallocated portions of the interpreter state. In order to jump to the peeled-iteration of a loop, the program state must be consistent with the state expected by the peeled-iteration, namely the state at the end of the preamble. The program state consists of optimization information such as the range of integer values, the state of heap objects, and which parts of the interpreter state exist virtually (due to allocation removal). In some cases, the JIT may introduce extra instructions add the end of the side trace to ensure the resulting state conforms with the desired peeled-iteration.

Program states often fail to match in Pycket, as different control flow paths examine different portions of the environment and current continuation, both of which are heap structures. Heavy use of impersonators exacerbates this problem.

To address the high failure rate of the loop peeling transformation, we improved the JIT's ability to force portions of the program state to match. In cases where a peeled-iteration expects an allocated heap object, the JIT may now allocate values that were elided by allocation removal. Rather than allocate the entire interpreter state, the JIT now allocates a small portion of the program state in exchange for jumping to an optimized peeled-iteration. If this strategy fails, the trace jumps to the preamble, and if this strategy succeeds, the JIT never allocates more than is required to jump to the preamble, so the new heuristic never adds overhead.

Across the entire benchmark suite, gradual typing overhead improves by 4% due to this optimization, and none of the benchmarks are negatively impacted. The `suffixtree` benchmark benefits most from this optimization, whose slowdown curve is shown in Figure 11. The untyped configuration of `suffixtree` suffered from a failure of the loop-peeling transformation, significantly degrading performance. Improvements to the loop-peeling transformation resulted in a 2× performance improvement. In order to prevent this from skewing the gradual typing overhead, figures comparing Pycket have their slowdowns computed relative to the full Pycket implementation.

The optimization described here is just one possible improvement which can be made to the JIT. The improvements from this optimization indicate there is still room for improvement by improving the JIT's ability to reason about code generated by Pycket for gradually typed programs.

5.5 Code Generation for Impersonators

This section will look at code generated by Pycket for a simple program which uses impersonators. Consider the code sample and corresponding JIT code in Figure 12. The Racket **for**-loop at the bottom of the figure iterates over a list of functions, invoking each on the value `0`. The result of each function call is added to a global accumulator to ensure the JIT does not elide the work in the loop. The JIT code in Figure 12 is generated for the body of the **for**-loop, iterating over `the-list`, a list of identity functions with three levels of impersonators.[9]. The handler for each procedure impersonator increments the input by one. With three layers, each list element is a computationally expensive way to write the function `(lambda (x) (+ x 3))`. If each function in the list is replaced by `(lambda (x) (+ x 3))`, Pycket's performance improves by 2×, while the Racket VM improves by 20×, 10× more overhead than Pycket.

The JIT code in Figure 12 occurs inline in the code generated for the body of the **for**-loop, the remainder of which (not shown for space) is responsible for deconstructing the list, checking the termination condition, and updating the mutable accumulator. At each impersonator level, the code checks that the current value is an impersonator (line 1), extracts the handler function and hidden class of the impersonator (lines 3 & 4), guards that these values are those used during code generation (lines 5 & 6), and then pulls out the next value in the impersonator chain (line 7). At the last level, the code ensures that the same underlying procedure is being called (line 21) and then performs a single operation which adds three to the accumulator.

```
1  (define (wrap f)
2    (impersonate-procedure f add1))
3
4  (define the-list
5    (for/list ([i (in-range 1000)])
6      (wrap (wrap (wrap (lambda (x) x))))))
7
8  (define total 0)
9  (for ([f (in-list the-list)])
10   (set! total (+ total (f 0))))
```

```
1  # examine outer impersonator
2  guard_class(p0, StructImpersonator)
3  p1 = getfield(p0, descr="handler")
4  p2 = getfield(p0, descr="hidden_class")
5  guard_value(p1, ConstPtr(ptr18))
6  guard_value(p2, ConstPtr(ptr16))
7  p3 = getfield(p0, descr="inner")
8  # examine middle impersonator
9  guard_class(p3, StructImpersonator)
10 p4 = getfield(p3, descr="handler")
11 p5 = getfield(p3, descr="hidden_class")
12 guard_value(p4, ConstPtr(ptr28))
13 guard_value(p5, ConstPtr(ptr26))
14 p6 = getfield(p3, descr="inner")
15 # examine inner impersonator
16 guard_class(p6, StructImpersonator)
17 p7 = getfield(p6, descr="handler")
18 p8 = getfield(p6, descr="hidden_class")
19 guard_value(p7, ConstPtr(ptr38))
20 guard_value(p8, ConstPtr(ptr36))
21 p9 = getfield(p6, descr="inner")
22 # check the underlying procedure
23 guard_value(p9, ConstPtr(ptr41))
24 # execute the underlying procedure's body
25 i11 = int_add_ovf(i10, 3)
26 guard_no_overflow()
```

Fig. 12. Simple Racket loop invoking a list of thrice impersonated identity functions. Each function is wrapped with three impersonators whose handler functions increment the input by one. Included is the JIT code generated for the invocation of the impersonated elements at `(f 0)`. The use of impersonators in this example has a 20× overhead in the Racket VM, but only 2× in Pycket.

The behavior of procedure impersonators is affected by the `application-mark` impersonator property. When the `application-mark` property is attached to an impersonator, the associated value is used to extend the current continuation mark set. The operations examining each hidden class ensure the absence of the `application-mark` property. These checks are cheap using hidden classes, the `guard_value` instructions are compiled to a pointer comparison and branch instruction. Without hidden classes, these checks become functions calls performing hash table lookups.

The Racket VM cannot produce such specialized code, which comes naturally from Pycket's use of RPython's meta-tracing JIT. The JIT code in Figure 12 interleaves Racket code and RPython code

[9]A list of functions is used to prevent the JIT from recognizing that each function is in fact loop invariant and hoisting all impersonator operations out of the loop

in Pycket's runtime. Unlike a traditional tracing JIT, the RPython JIT is able to optimize across the user code/runtime boundary, requiring little additional work to add JIT support for impersonators.

In contrast, the Racket VM's JIT performs few optimizations in the presence of impersonators. The Racket VM calls into the C runtime to handle operations on impersonated values, occasionally producing JIT code to directly call handler functions (a dated account is given by Strickland et al. [2012]).

The use of impersonators to enforce soundness means that when and where runtime checks occur is highly control/data flow dependent. A key insight into Pycket's ability to optimize gradually typed programs is that Pycket generates specialized code by observing rather than predicting execution, including the handling impersonators, generating specialized code based on the impersonator structures encountered at runtime. This allows the JIT to seamlessly optimize the code packaged in an impersonator with the context in which it is invoked.

6 THREATS TO VALIDITY

Applicability to other JIT compilers. Pycket's ability to optimize gradually typed programs stems partially from the use of trace compilation. Tracing (and meta-tracing in particular) naturally produces the optimized code paths needed to efficiently implement contracts mediated by proxies. Trace compilation is not as widely used as per-function JIT compilation. JavaScript, a language for which several gradual type systems have been developed [Rastogi et al. 2015; Richards et al. 2015], no longer makes use of trace compilation in any widely used implementation.

Extending our results to more run-of-the-mill per-function JIT compilers is an open problem. The key to Pycket's success is not tracing, in and of itself, but the aggressive runtime feedback during compilation which naturally comes from tracing. Accomplishing the same feedback in a per-function JIT usually requires a separate, carefully instrumented profiling mechanism [Hackett and Guo 2012].

Macro-level Gradual Typing. Another concern is Typed Racket's adherence to macro-level gradual typing. The course grained nature of macro-level gradual typing makes the overheads of gradual typing more tractable by reducing the number of typing boundaries in a program. The additional overheads of micro-level gradual typing are a matter of degree rather than kind. While Typed Racket does not allow for direct experimentation with micro-level gradual typing, Racket's contract system allows for the fine grained application of contracts, similar to micro-level gradual typing. Pycket still improves the performance of such programs, though it is unknown whether the performance micro-level gradual types is amenable to real world use under Pycket.

Evaluation Metric. Finally, the use of "slowdown" as the appropriate metric is predicated on a number of assumptions. Most glaringly, slowdown assumes that the overhead is a constant factor with respect to the amount of work performed by benchmark. Typed Racket does not guarantee space efficiency for its implementation of gradual typing, though much work has gone into ensuring it for common cases. Indeed, it is possible to produce programs whose asymptotic behavior is negatively impacted by the introduction of gradual types (the **Binomial** benchmark in Figure 8 is an example using only the contract system).

The kinds of contracts generated by Typed Racket allow for space efficiency gains which are not available to general contracts, but there are currently no guarantees that a program's asymptotic behavior is not altered by the use of gradual types. Asymptotic overheads are purely the fault of Typed Racket and cannot be alleviated by Pycket alone.

7 RELATED WORK

Little work has been done on optimizing sound gradual typing from the ground up. Solutions to poor performance focus on changing the language semantics, rather than adapting the execution environment to the unique challenges of sound gradual typing. We survey those solutions here, as well as previous measurements of gradual typing performance.

The authors of Gradualtalk [Allende et al. 2014] outline several lines of research on optimizing gradual typing in a position paper [Allende and Fabry 2011]. Therein, they note the difficulty of optimizing gradually typed programs statically; in particular, they note that the boundary between static and dynamic code depends on the control flow of the program, rather than static properties of the source code. Trace compilation, as in Pycket, discovers these boundaries during tracing and optimizes across them.

One well studied aspect of gradual typing performance is where to do dynamic checks—sometimes referred to as the "cast insertion strategy" [Allende et al. 2013; Vitousek et al. 2014]. It is common for the semantics of a gradually typed language to be specified by translation to a language with casts. Casts perform the runtime checks which ensure soundness of the gradual type system. Allende et al. [2013] examine the effects of cast insertion strategy on runtime and memory overhead in Gradualtalk. Their results show significant runtime improvements for their hybrid strategy, which moves some checks into the bodies of typed methods (at the cost of a larger memory footprint). However, this system uniformly *adds* overhead to the untyped performance, rather than enabling any performance improvements.

In a similar vein, Vitousek et al. [2014] detail three possible casting strategies for Reticulated Python (a gradually typed variant of Python). Their `guarded` strategy makes use of proxies in a manner similar to Typed Racket, and has substantial performance overhead. In contrast, their `transient` strategy adds additional checks in the typed code to ensure typing assumptions have not been violated for values which flow into untyped code. This strategy has other benefits in Reticulated, but does not preserve blame or catch errors as early, relative to `guarded` or Typed Racket. Both strategies impose overhead in the statically typed code, though the `transient` strategy is measurably faster. Vitousek et al. [2014] also discuss but do not evaluate a third strategy, `monotonic` [Siek et al. 2015], which signals *more* errors, but avoids many proxy wrappers. To date, this strategy has not yet been empirically evaluated.[10]

The performance of the `transient` strategy was measured by Vitousek et al. [2017]. For many programs, the overhead of the dynamic checks is small in the absence of blame tracking. While the cost of gradual typing exceeds 5× for some benchmarks, this slowdown is much smaller than the cost of their blame tracking strategy.

Thorn [Wrigstad et al. 2010] takes a different approach to gradual typing, introducing the concept of `like` types. Every type `C` has a corresponding `like C` type, whose *uses* are checked against `C`'s interface statically, but need not be statically known to have type `C`. Values accessed though `like` typed variables are checked dynamically to ensure they do indeed conform to the expected interface. Thus, `like` types provide a middle ground between fully static and fully dynamic types. Wrigstad et al. [2010] provide some limited benchmarks demonstrating significant performance gains by fully statically typing their benchmarks. They also note that Thorn is able to produce more efficient bytecode sequences for functions expecting `like` types, however, the evaluation provides timing results only for the untyped and typed configurations of the benchmark, so the exact improvements are unclear. Furthermore, the performance comparison does not include any sophisticated JIT or ahead-of-time compilers, making comparison to Pycket difficult.

[10]Personal communication with the authors of Reticulated Python.

Safe TypeScript [Rastogi et al. 2015] (a sound variant of TypeScript [Bierman et al. 2014]) reports significant overhead (2.4–72× slowdown) without type annotations, though performance is mostly recovered once the program is fully typed. The overhead on untyped code is due to wrappers inserted to enforce security isolation properties. Their performance analysis is performed on top of Google's V8 JavaScript engine, a sophisticated, method-based JIT compiler able to optimize code based on runtime type profiling [Millikin and Schneider 2010]. Though performance on intermediate states in the gradual typing lattice are not reported, they report that Safe TypeScript averages 22× slowdown for the fully untyped configurations, far more than Pycket's worst performance on any configuration.

Similarly, SoundScript [Richards et al. 2015] adapts the Thorn approach to TypeScript, dubbing the soundly-checked types "concrete types." They report on performance improvements when programs are typed, using a modified version of the V8 virtual machine. However, StrongScript omits both blame tracking and structural types, the key features that impose performance overhead in Typed Racket. The authors write "we decided to remove [blame tracking] as it did incur serious performance overheads". Their evaluation did not measure partially-typed programs and thus does not examine the overhead of gradual typing.

TypeScript, and thus both SoundScript and Safe TypeScript, as well as Reticulated Python offers micro-level gradual typing, whereas Typed Racket provides macro-level gradual typing. Micro-level gradual typing presents further performance challenges, as the fine granularity of type annotations increases the potential number of typed/untyped boundaries in the resulting program. It remains an open question whether Pycket (or a similar system) could bring the overhead of a micro-level sound gradual typing system to acceptable levels.

8 CONCLUSION

Sound gradual typing is a desirable feature in a modern programming language, but it comes at a cost. The overhead of enforcing static types at runtime is unacceptable for conventional language implementations. This fact is well noted in the literature, though little work has been done to systematically reduce this overhead, until now. The recent work of Takikawa et al. [2016] warns that unless performance improves significantly, sound gradual typing is effectively "dead."

We present Pycket, a Racket implementation using a state of the art meta-tracing JIT compiler, which offers the required performance improvement. These improvements rely on the tracing JIT, to which we add as well as on a new runtime representation of proxies using hidden classes which reduces both time and space overhead. The benchmarking results reported in Section 4 show that gradual typing overheads can be brought within acceptable bounds for real world applications, for a gradual typing system with a significant user base, by using a sophisticated JIT compiler.

Our system brings the performance of sound gradual typing into the realm of practicality for many real world applications: 45% of all configurations suffer a slowdown of no more than 10% and 66% are within one translation step from a 10% slowdown. This is not the final word in gradual typing performance—there is still room for significant improvement—but a demonstration of tractability. Pycket represents a significant step in the efficient execution of gradually typed programs. In light of this, we jest that sound gradual typing is "only mostly dead." [Reiner 1987]

ACKNOWLEDGMENTS

Bolz-Tereick is supported by the EPSRC *Cooler* grant EP/K01790X/1. Siek and Tobin-Hochstadt are supported by NSF Grant 1518844.

This research was supported in part by Lilly Endowment, Inc., through its support for the Indiana University Pervasive Technology Institute, and in part by the Indiana METACyt Initiative. The Indiana METACyt Initiative at IU is also supported in part by Lilly Endowment, Inc.

Robby Findler and Matthew Flatt contributed performance improvements to Racket and Typed Racket.

We would also like to thank Miracle Max as a constant source of inspiration through the course of this work.

REFERENCES

Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. 1986. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction (SIGPLAN '86).* ACM, 219–233.

Esteban Allende, Oscar Callau, Johan Fabry, Éric Tanter, and Marcus Denker. 2014. Gradual typing for Smalltalk. *Science of Computer Programming* 96 (2014), 52–69.

Esteban Allende and Johan Fabry. 2011. Application optimization when using gradual typing. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems.* ACM, 3.

Esteban Allende, Johan Fabry, and Éric Tanter. 2013. Cast Insertion Strategies for Gradually-typed Objects. In *Proceedings of the 9th Symposium on Dynamic Languages (DLS '13).* ACM, 27–36.

Håkan Ardö, Carl Friedrich Bolz, and Maciej Fijałkowski. 2012. Loop-aware Optimizations in PyPy's Tracing JIT. In *Proceedings of the 8th Symposium on Dynamic Languages (DLS '12).* ACM, New York, NY, USA, 63–72.

Edd Barrett, Carl Friedrich Bolz, Rebecca Killick, Vincent Knight, Sarah Mount, and Laurence Tratt. 2016. Virtual machine warmup blows hot and cold. *arXiv preprint arXiv:1602.00602* (2016).

Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G Siek, and Sam Tobin-Hochstadt. 2015. Pycket: A tracing JIT for a functional language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming.* ACM, 22–34.

Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014–Object-Oriented Programming.* Springer, 257–281.

Gavin Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding Dynamic Types to C#. In *European Conference on Object-Oriented Programming.* Springer, 76–100.

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijakowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. 2011. Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages. In *Proc. ICOOOLPS.* 9:19:8.

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijakowski, and Armin Rigo. 2009. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proc. ICOOOLPS.* 18–25.

Carl Friedrich Bolz, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. 2014. Meta-tracing makes a fast Racket. In *Workshop on Dynamic Languages and Applications.*

Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. 2016. Practical Optional Types for Clojure. *European Symposium on Programming* 25 (2016).

C. Chambers, D. Ungar, and E. Lee. 1989. An Efficient Implementation of SELF a Dynamically-typed Object-oriented Language Based on Prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA '89).* ACM, 49–70.

Olivier Danvy. 1991. *Three Steps for the CPS Transformation.* Technical Report CIS-92-02. Kansas State University.

Facebook. 2017a. Flow Documentation. (2017). http://flowtype.org/docs/about-flow.html

Facebook. 2017b. Hack Documentation. (2017). https://docs.hhvm.com/hack/

Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine and the lambda-calculus. In *Working Conf. on Formal Description of Programming Concepts - III.* Elsevier, 193–217.

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

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. In *Proc. PLDI.* 502–514.

Google. 2015. *Dart Programming Language Specification.* Technical Report. EMCA International. https://www.dartlang.org/docs/spec/

Brian Hackett and Shu-yu Guo. 2012. Fast and Precise Hybrid Type Inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12).* ACM, New York, NY, USA, 239–250.

Jukka Lehtosalo and David J Greaves. 2011. Language with a Pluggable Type System and Optional Runtime Monitoring of Type Errors. In *Proceedings of International Workshop on Scripts to Programs (STOP).*

André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. 2014. Typed lua: An optional type system for lua. In *Proceedings of the Workshop on Dynamic Languages and Applications.* ACM, 1–10.

Kevin Millikin and Florian Schneider. 2010. A New Crankshaft for V8. *The Chromium Blog* 7 (2010).

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, 167–180.

Rob Reiner. 1987. The Princess Bride. (1987).

Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete types for TypeScript. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Armin Rigo and Samuele Pedroni. 2006. PyPy's Approach to Virtual Machine Construction. In *Proc. DLS*. 944–953.

Thomas Schilling. 2013. *Trace-based Just-in-time Compilation for Lazy Functional Programming Languages*. Ph.D. Dissertation. University of Kent.

Jeremy G Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.

Jeremy G Siek, Michael M Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015. Monotonic references for efficient gradual typing. In *Programming Languages and Systems*. Springer, 432–456.

T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. Chaperones and impersonators: run-time support for reasonable interposition. In *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 *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S New, Jan Vitek, and Matthias Felleisen. 2016. Is sound gradual typing dead?. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 456–468.

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage migration: from scripts to programs. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 964–974.

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 395–406.

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI '11)*. ACM, 132–141.

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS '14)*. ACM, 45–56.

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 *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 762–774.

Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. Integrating Typed and Untyped Code in a Scripting Language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, 377–388.