Towards Practical Gradual Typing*

Asumu Takikawa[1], Daniel Feltey[1], Earl Dean[2], Matthew Flatt[3], Robert Bruce Findler[4], Sam Tobin-Hochstadt[2], and Matthias Felleisen[1]

1 **Northeastern University**
Boston, Massachusetts, USA
{asumu,dfeltey,matthias}@ccs.neu.edu
2 **Indiana University Bloomington**
Indiana, USA
{samth,edean}@cs.indiana.edu
3 **University of Utah**
Salt Lake City, Utah, USA
mflatt@cs.utah.edu
4 **Northwestern University**
Evanston, Illinois, USA
robby@eecs.northwestern.edu

Abstract

Over the past 20 years, programmers have embraced dynamically-typed programming languages. By now, they have also come to realize that programs in these languages lack reliable type information for software engineering purposes. Gradual typing addresses this problem; it empowers programmers to annotate an existing system with sound type information on a piecemeal basis. This paper presents an implementation of a gradual type system for a full-featured class-based language as well as a novel performance evaluation framework for gradual typing.

1998 ACM Subject Classification D.3 Programming Languages

Keywords and phrases Gradual typing, object-oriented programming, performance evaluation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.4

1 Gradual Typing for Classes

Gradual type systems allow programmers to add type information to software systems in dynamically typed languages on an incremental basis [39, 48]. The ethos of gradual typing takes for granted that programmers choose dynamic languages for creating software, but also that for many software engineering tasks, having reliable type information is an advantage. The landscape of gradual typing includes many theoretical designs [26, 29, 39, 40, 46, 53], some research implementations cite3,20,49,52,55, and, recently, the first industrial systems (Typescript [51], Hack[1], Flow[2]).

Despite these numerous efforts, no existing project deals with the full power of object-oriented programming in untyped languages, e.g., JavaScript, Python, Racket, Ruby, or

* Due to a conflict of interest, we could not submit an official artifact for consideration to the ECOOP Artifact Evaluation Committee. However, we have prepared an unofficial artifact that is available at the following URL: http://www.ccs.neu.edu/home/asumu/artifacts/ecoop-2015.tar.bz2.

[1] See http://hacklang.org/ and Verlaguet, *Commercial Users of Functional Programming*, Boston, MA 2013.

[2] See http://flowtype.org.

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 4–27

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
(define C1
  (class object% (super-new)
    (define/public (m) "c1")))
(define C2
  (class object% (super-new)
    (define/public (m) "c2")))
; f is a mixin, result inherits from C
(define (f C)
  (class C (super-new)
    (define/public (n)
      (send this m))))
(define-values (C1* C2*)
  (values (f C1) (f C2)))
```

```
class C1(object):
  def __init__(self): pass
  def m(self): return "c1"
class C2(object):
  def __init__(self): pass
  def m(self): return "c2"
# f is a mixin, result inherits from C
def f(C):
  class Sub(C):
    def __init__(self): pass
    def n(self): return self.m()
  return Sub
c1cls, c2cls = f(C1), f(C2)
```

Figure 1 First-class classes in both Racket (left) and Python (right).

Smalltalk. Among other things, such languages come with reflective operations on classes or classes as first-class run-time values. See figure 1 for an abstracted example of functions operating on classes.[3] While users of dynamically-typed languages embrace this flexibility and develop matching programming idioms, these linguistic features challenge the designers of gradual type systems. Only highly experimental languages in the statically typed world [34] support such operations on classes, and only our previous theoretical design [46] deals with the problem of how to turn such a type system into a sound gradual type system.

This paper presents the first *implementation* of a sound gradual type system for a higher-order, class-based, practical OO language that supports runtime class composition.? Abstractly, it makes three concrete contributions: (1) design principles for gradual typing in a higher-order, object-oriented context, (2) a new mechanism to make the soundness-ensuring contract system performant, and (3) a novel performance analysis framework for gradual type systems. Our project is based on the Racket language [18], because it comes with a typical untyped class system [16] and a gradual type? system [49] implemented as a library [50]. Furthermore, our previous theoretical design [46] was made with Typed Racket in mind.?

Section 2 describes the design and its base principles, while sections 3 and 4 explain the evaluation. Section 5 presents related work; Section 6 suggests general lessons.

2 The Design Space

The design of object-oriented Typed Racket (TR) is informed by basic principles and a formative[4] evaluation. The first subsection explains the principles. The second illustrates them with a series of examples, focusing on how TR accommodates existing untyped idioms. The remainder sketches the challenges, solutions, and limitations.

2.1 By Principle

The design principles of TR fall into two categories: those derived from our previous work on a gradually typed calculus of first-class classes [46] and our practical implementation

[3] We thank Laurence Tratt for inspiring the Python version.
[4] We borrow the terms “formative evaluation” and “summative evaluation” from the education community [36]. A *formative evaluation* informs the design process; a *summative evaluation* assesses its outcome.

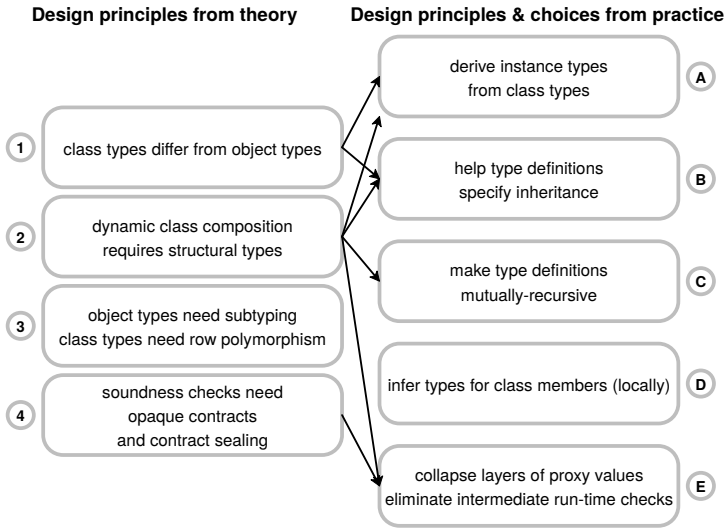

Figure 2 Design principles for Typed OO Racket.

experiences. Figure 2 presents these principles in two columns that correspond to the two categories: the theoretical principles on the left and their induced implementation concerns on the right.

Theoretical principles (1–4). Racket supports classes as first-class values, giving rise to design patterns such as mixins and traits [16]. Class expressions produce class values that freely flow through the program, including functions and data structures:

```
; several class values used at run-time
(define (make-class base-class)
  (class base-class
    (super-new)
    (define/public (new-method arg) ....)))
(define c% (make-class object%))
(list c% c% c%)
```

The first line shows a function that maps a class to a subclass. Such functions are *mixins* because they add new points to an existing class hierarchy. The second line shows a use of the mixin, the third one a list of three class values.

Since class values may appear anywhere and without a statically-known name, we cannot simply identify classes by their names as in *nominal type systems* such as those of Java or C#. Instead we introduce class types that are compared *structurally*. Put differently, classes are not identified just by a name, but by the types of their members (i.e., methods, fields, etc.). This matches up with principle (2) in figure 2; it is also unsurprising as other researchers have proposed a similar approach [3, 5, 8, 33, 52].

Furthermore, the type system must distinguish the types of classes from the types of objects, because expressions can yield either of these; see principle 1. In addition, class types must support polymorphism because mixins are parametrized over base classes. To accommodate this idiom, TR's type system borrows *row polymorphism* from the extensible record and object literature; see principle 3.

By (3), the type system restricts row polymorphism to class types and disallows *width subtyping*; allowing both is unsound [46]. Conversely, it accommodates existing design patterns via object types that have width subtyping but lack row polymorphism.

Unlike an ordinary type system, a *gradual* type system must also support the sound interoperation of typed and untyped code. In TR, *higher-order contracts* mediate these interactions. Concretely, when a typed module in a gradually-typed language exports to/imports from an untyped one, the type system must translate any type specifications into contracts that uphold the type's invariants at run-time. On top of ordinary class contracts [44], principle (4) requires *sealing* contracts for protecting mixins and *opaque* contracts that prevent untyped programs from accessing hidden methods on objects. These contracts are applied to the actual class values that flow to untyped code, and thus the untyped code always interacts with typed values through a protective wrapper that identifies type violations by *blaming* the responsible untyped code.

Practical principles (A–E). Principles based on a small calculus [46] never suffice for a real-world design. In the process of creating TR's gradual type system, five additional design concepts emerged, which we consider as fundamental as the theoretical ones.

The separation of class types from object types calls for a linguistic mechanism that derives types of instances from types of classes or vice versa. The key to this syntax design is to choose a convenient default choice. From this perspective, it is important that instance types are easily synthesizeable from a class type while the type of an instance lacks information – e.g., the constructor – that is needed to reconstruct the type of the class. This insight naturally leads to a choice that makes class types the primary vehicle and introduces an `Instance` type constructor for the derivation of object types from class types.

While a gradual type system for dynamic classes demands the introduction of structural types, writing down such types imposes a significant notational overhead. In practice, class definitions consist of nests of deeply interwoven "has-a" or "is-a" relationships. For example, one class type from TR's standard GUI library consists of 245 methods signatures, and moreover, the type refers to instances of 23 other class types plus itself. Not surprisingly, some of these 23 class types refer back to the original class type. In short, a gradual typing system for a dynamic object-oriented language needs a powerful construct for defining large mutually recursive types.

To accommodate sharing of features in class types, TR's `Class` type constructor comes with an optional `#:implements` clause that allows a given class type to copy another class type's member definitions. Principle (B) captures this point.

One way to accommodate nests of self-recursive and mutually-recursive type definitions is to encode them with ordinary μ-types [11], which already exist in functional TR. Simple experiments in writing types for our GUI library expose the drawbacks of this approach. Therefore TR instead provides `define-type`, a novel[5] form of mutually-recursive structural type definitions. It makes up for the lack of type recursion through class names that nominally-typed language such as Java get for free. These named class types are *not* nominal as in Java because the equivalence between two type definitions is always determined by structural comparison and recursive unfolding. At the same time, our type definitions retain the simple syntax of type aliasing.

[5] Superficially, these mutually recursive type definitions resemble OCaml's mutually recursive classes, but the two differ starkly. Our design separates the class type definitions from the definitions of the actual classes and allows semantic forward declarations without imposing syntactic ones.

Real-world programming also means reducing the number of *required* type annotations. A gradual type system therefore needs an algorithm that reconstructs some types automatically (principle D). Module-level type inference à la Hindley & Milner causes too many problems, however. Hence, TR *locally* infers the types of class members when possible, e.g., fields initialized with literal constants.

Finally, by its very existence, the *theoretical* work [21, 25, 41] on collapsing higher-order run-time checks implicitly conjectures that layering proxy wrappers around values causes a degradation in the performance of gradually typed programs. In contrast, the extensive use of *functional* Typed Racket over six years in open-source and industrial projects had up to now not provided *practical* evidence for this prediction. The addition of object-oriented features to Typed Racket reveals that the possibility is indeed real. The naïve extension of Typed Racket suffers from exponential growth in object wrappers on real-world examples. It is likely that higher-order object-oriented programs are prone to these kinds of interactions due to the widely used model-view architecture. In any case, our work finally demonstrates the practical need of optimizing for space usage.

Constructively put, we articulate principle (E). All implementations of sound gradual type systems with structurally typed classes need a way to merge layers of proxy wrappers. One such implementation in the literature, Reticulated Python [52], strictly follows Siek and Wadler's theoretical proposal of collapsing casts into "threesomes" [41]. Typed Racket employs an alternative solution, which is sketched in section 4.3.

2.2 By Example

To provide evidence that our design ideas from the previous section scale to real code, we walk through a series of examples extracted from our case studies described in section 3. In particular, the examples demonstrate how to add types to untyped code, how typed and untyped code interact, and how the type and contract systems can handle untyped idioms.

Augmenting an existing codebase. Recall that the gradual-typing thesis states that a maintenance programmer ought to be able to augment an existing untyped code base with types in order to benefit from their software engineering advantages, and to ensure that future programmers will continue to receive those benefits. Our illustration starts with an excerpt from the Racket Esquire program in figure 3 [15], which implements a bare-bones interactive editor combined with a Lisp-style Read-Eval-Print-Loop. The excerpt showcases the definition of a typical mixin.

This particular mixin, named `esq-mixin`, adds REPL capabilities to a base text editing class, such as the `text%` class from the GUI standard library on the last line.

The body of the mixin uses `class` to derive a subclass from `base-class`, the function's parameter. The rest of the class form contains typical elements of object-oriented programming: a call to a superclass constructor, several private fields, public method definitions, and an overriding method definition. As in C#, overriding methods in Racket are explicitly signaled with a `define/override` keyword. The `inherit` form both ensures that the superclass contains the given method names and allows the given superclass methods to be called without explicit `super` calls. The call to `new-prompt` in the class body executes when an instance of the class is constructed. In general, any expressions in the class body are run as part of the class's instantiation process.

Our code snippet in figure 3 calls for two pieces of type information: the mixin's argument and the public methods in the mixin's result. Furthermore, since the mixin is polymorphic,

```
(define (esq-mixin base-class)
  (class base-class
    (super-new) ; run the superclass initialization
    ; inherit methods for use from the superclass
    (inherit insert last-position get-text erase)
    (define prompt-pos 0) (define locked? #t) ; private fields
    (define/public (new-prompt)
      (queue-output (λ () (set! locked? #f)
                          (insert "> ")
                          (set! prompt-pos (last-position)))))
    (define/public (output str)
      (queue-output (λ () (let ([was-locked? locked?])
                            (set! locked? #f)
                            (insert str)
                            (set! locked? was-locked?)))))
    (define/public (reset)
      (set! locked? #f) (set! prompt-pos 0)
      (erase)
      (new-prompt))
    (define/override (on-char c)
      (super on-char c)
      (when (and (equal? (send c get-key-code) #\return) (not locked?))
        (set! locked? #t)
        (evaluate (get-text prompt-pos (last-position)))))
    (new-prompt))) ; method call during class initialization
(define esq-text% (esq-mixin text%)) ; application of the mixin
```

Figure 3 Racket Esquire, untyped.

we explicitly specify a row variable for the mixin's type. Figure 4 shows the mixin with a type annotation that encodes the required information.

The top of figure 4 displays a definition for `Esq-Text%`, the type for `esq-text%`. We also use it for the type annotation on `esq-mixin`. We take advantage of the `Class` constructor's built-in support for type inheritance to make `Esq-Text%` inherit from the `Text%` type defined in TR's base type environment.

Since `esq-mixin` is a row polymorphic function, we annotate it with the `All` and `->` type constructors. Given the two type definitions `Text%` and `Esq-Text%`, we can describe the domain and result types of the function type. Both `Class` types specify two key pieces: the types are row polymorphic due to the `#:row-var r` clause and the types inherit from existing structural types `Text%` and `Esq-Text%`, respectively. The former indicates that the class contains an unspecified set of additional methods or fields that are determined when the mixin is actually applied.

The method types inherited from `Text%` and `Esq-Text%` are used to type-check the body of the mixin. For example, the types of inherited methods such as `insert` are deduced from `Text%`. The types on the public methods are given in the definition of `Esq-Text%`. The types for the public methods document the methods' arguments and their effectful nature.

In this example, Typed Racket requires a single annotation to type-check the mixin, which reflects principle E from figure 2 about reducing the burden on the maintenance programmer. In particular, none of the private members or inherited fields need annotations.

```
; in module Library
(define-type Esq-Text%
  (Class #:implements Text%
         [new-prompt (-> Void)]
         [output (String -> Void)]
         [reset (-> Void)]))

(: esq-mixin (All (r #:row)
                  (-> (Class #:row-var r #:implements Text%)
                      (Class #:row-var r #:implements Esq-Text%))))
(define (esq-mixin base-class) (class base-class ....)) ; as before
```

Figure 4 Racket Esquire, typed.

```
(Class #:implements Text%
       [new-prompt (-> Void)]
       [output (String -> Void)]
       [reset (-> Void)])
```

```
(class/c ; many cases elided from Text%
         [new-prompt (-> void?)]
         [output (-> string? void?)]
         [reset (-> void?)])
```

Figure 5 Translating a type to a contract.

Cooperating with untyped code. Imagine that `esq-text%` is integrated into a complete project where other code – say an IDE system – remains untyped. In order to ensure that the widget's type invariants are upheld, the system must dynamically check that the untyped code uses the widget safely.

Concretely, consider a program divided into Library and Client modules. The Library module provides the Esquire text editing functionality, while the Client module uses it as part of a larger program. Since the Client imports the `esq-text%` class, the class value itself flows from the Library, passing through a boundary between the typed world and untyped world on the way.

Now consider a concrete snippet from the Client module:

```
; in module Client
(require "library.rkt")
(define repl-text (new esq-text%))
(send repl-text output 42)
```

The method invocation with **send** clearly violates the `String` type specified on the `output` method. Although the type-checker would catch such a mistake for the Library module, it is unable to inspect the untyped Client code.

Instead, the type-checker translates the type to a contract that ensures the type invariants. Figure 5 shows the result of (automatically) translating the type for the Esquire class to a matching class contract. As explained in section 2.1, these class contracts are opaque, meaning they disallow the export of a class with methods not explicitly listed in the type.

When classes flow from untyped modules to typed modules, the typed code must specify types for these imports. Suppose that we need to import the `text%` class directly from the standard, untyped GUI library. Assuming `Text%` is defined, the typed portion could use an import specification like this:

```
(require/typed racket/gui
               [text% Text%])
```

The `require/typed` form in TR imports the given bindings with the given type specifications. As before, these types are translated to contracts, ensuring that the imports live up to the desired specifications.

Mixins and typed-untyped interoperation. Let us illustrate the typed-untyped interoperation with an example of a mixin from the Big-Bang event-based functional I/O library. The library uses mixin methods such as these:

```
; if no callbacks are provided (on-key, on-pad, on-release), don't mix in
(define/public (deal-with-key base-class)
  (if (and (not on-key) (not on-pad) (not on-release))
      base-class
      (class base-class
        (super-new)
        ; the method invokes the callbacks supplied by the user
        (define/override (on-char e) ...))))
```

This method accepts a class argument named `base-class` and, when appropriate, returns a subclass that adds a custom key event handler. The method's implicit precondition requires that the class `base-class` already contains a `on-char` method to be overridden. The remaining members of the class are unconstrained.

To import a class with such method, a programmer may write

```
(require/typed
 [world% (Class ...
                [deal-with-key
                 (All (r #:row) ; method types elided for space
                   (-> (Class #:row-var r [on-char ...])
                       (Class #:row-var r [on-char ...])))])])
```

The (automatic) translation of a row-polymorphic function type into a contract requires a seal [46] for the `deal-with-key` method:

```
(sealing->/c (X) [on-char]
   (and/c X (class/c [on-char ...]))
   (and/c X (class/c [on-char ...])))
```

The `sealing->/c` combinator creates a function contract that generates a fresh class seal when the wrapped function is applied. The occurrences of `X` in the body of the combinator are replaced at run-time with either a sealing or unsealing operation depending on whether the variable occurs in a negative or positive position. Each seal lists those class members that are left unsealed (here, the `on-char` method); all unmentioned members are hidden. Thus the method contract above explicitly seals off all members from the argument class `base-class` in the implementation except `on-char`. This prevents the mixin from adding or overriding any methods other than `on-char`, which matches the polymorphic type and expresses the intent of the method in a precise manner.

Examples of mutually-recursive type definitions. The complex relationships between classes in practice requires the use of mutually-recursive type definitions, see figure 2 (C). The type for the `text%` class is highly illustrative:

```
(define-type Text-Object (Instance Text%))
(define-type Text%
  (Class ; 244 other methods ...
         [set-clickback
          (Natural Natural (Text-Object Natural Natural -> Any)
           -> Void)]))
```

The `set-clickback` method for `text%` objects installs a callback triggered on mouse clicks for a region of the text buffer. The type for that callback function recursively refers to the `Text%` type via the `Text-Object` definition. No explicit recursive type constructors are necessary to write the type down. Under the covers, TR establishes the recursion through the type environment, even though `Text%` is not a nominal type.

2.3 From Types to Contracts: An Implementation Challenge

Soundness calls for run-time checks that enforce the type specifications when a value flows from the typed portion of a program to an untyped one. All theoretical designs choose either casts or contracts for this purpose. GradualTalk [3] and Reticulated Python [52] rely on the former; research in this realm focuses on what kind of casts to implement and how this choice affects the expressiveness of the language and the efficiency of programs. In contrast, TR is the first implementation of a gradually typed, object-oriented languages that uses higher-order contracts instead of casts.

Although the homomorphic translation from types to contracts (illustrated above) is easy to use in theory work [46], implementing it for practical purposes is not easy. For the kinds of structural types used in the TR code base, it generates excessively large contracts. To achieve reasonable results, it is critical to treat the problem of translating types to contracts as a compilation problem that requires an optimization phase.

Based on several failed attempts, TR now compiles a recursive type to a recursive declaration of mutually recursive contracts for each dependency. To create contracts of manageable size, the analysis of the dependencies among the type definitions finds cycles in the dependency graph. Definitions within these cycles are lifted and memoized. Definitions that do not participate in cycles incur no overhead.

2.4 Limitations

TR currently suffers from a few limitations, which include some workarounds for our case studies, but rarely prevent the conversion of an untyped module.

Row polymorphism on objects. Our system provides row polymorphic types for supporting mixins, but only class types are allowed to contain row variables. Unlike most designs with row polymorphism such as OCaml, the types for objects are concrete and support standard width subtyping. This tradeoff works well for most of our examples, because Racket programmers are often content with Java-style use of classes. Our choice rules out row polymorphic functions that construct an object from a given class or the use of row polymorphism to emulate bounded polymorphism for objects. In practice, the lack of row polymorphism for objects prevents us from porting a *single* module in the DrRacket IDE. We conjecture that adding bounded polymorphism to Typed Racket would fill this gap.

Occurrence typing for OO code. One of the important features of functional TR is its use of *occurrence typing* [49]. To accommodate dynamic type-tag tests, TR refines the types

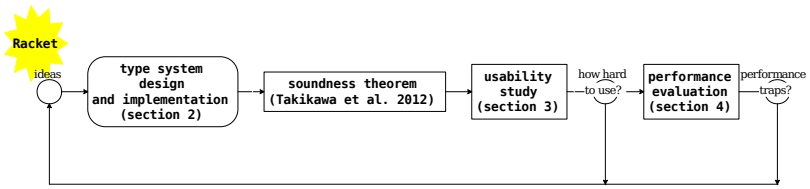

Figure 6 The feedback cycle for Typed OO Racket.

of variables depending on where they occur. If, say, a dynamic test checks whether `x` is a non-empty list or `y` is a number in a specific interval, their types in the `then` and `else` branches of a conditional reflect the possible results of these checks. Occurrence typing is crucial for porting untyped programs into the typed world because the former often discriminates elements from unions of data via predicates.

Sadly, while TR supports occurrence typing on private class fields, it cannot support two important uses for OO constructs: (1) recovering object types from uses of `is-a?`, which is like `instanceof` in Java, and (2) occurrence typing on public fields. For an example of the first, suppose we export the typed `esq-text%` class from earlier to untyped code. If we encounter the test `(is-a? an-object esq-text%)` in typed code, we would like to conclude that the value `an-object` has the type `(Instance Esq-Text%)`. Unfortunately, this is unsound if `an-object` originates in untyped code, since the untyped code may have subclassed `esq-text%`, overridden its methods with ill-typed implementations, and constructed `an-object` from that subclass. Closing this gap requires additional research.

For the second problem, consider how a concurrent thread may mutate a public field between a tag check and the execution of the corresponding branch of the conditional. An application of occurrence typing could then lead to an incorrect type for a field based on an out-of-date tag check. We therefore will investigate immutable public fields in the future.

3 Effectiveness Evaluation

Like all good design efforts, our design of Typed Racket takes place in the context of a feedback cycle. Figure 6 visualizes our particular feedback loop. With respect to this paper, two elements stand out: the usability study and the performance evaluation. This section presents the former for Typed OO Racket and its design impact. The next section introduces a novel performance evaluation framework and discusses how it influenced the design.

The usability evaluation aims to test three hypotheses:[6]

1. Typed Racket enables programmers to add types in an incremental manner, including for components that dynamically create and compose classes. This hypothesis demands two specific qualities from the type annotation process: the burden of adding type annotations must be small, and the program logic should rarely change.
2. Each theoretical design principle (1–4) is needed for realistic programs.
3. Each practical design principle (A–E) helps annotate realistic programs.

[6] Our goal is *not* to determine whether static typing per se contributes to software maintenance, deferring instead to the existing literature [22].

3.1 Cases

The evaluation was conducted in two stages: a formative evaluation and a summative one. For the summative evaluation, the programmer had no prior knowledge of Racket's class system or Typed Racket's gradual type system.

With two exceptions, the code bases in this section come from the mature Racket distribution. The newest dates from Racket version 5.3.2, released in January 2013; the rest have been shipped in user-tested distributions for a minimum of two years and some for nearly twenty years. The two exceptions are Acquire and Esquire.

Our *formative* evaluation employed four case studies: a tool that inserts Large Letters into a program text, a Tooltip drawing library, the functional Big-Bang I/O library [12], and Esquire. The Big-Bang case study is about annotating the library's graphical core while leaving the remaining pieces untyped. Finally, Esquire is a graphical REPL that illustrates the essential elements of DrRacket; some of the code snippets in section 2 are taken from Esquire.

The nine cases included in the *summative* evaluation cover the full range of object-oriented programming idioms in Racket. Several cases are extracted from a package of games that are included in the Racket distribution: Mines, a graphical Minesweeper game, Slidey, a puzzle game, Cards, a library for the standard 52-card deck, and Aces and GoFish, two games using Cards. The Markdown component is one of several renderers for the Scribble documentation language [14]. The DB case study covers a library that provides access to SQLite databases. Finally, the Acquire board game is a project from a programming course that represents an interactive system with a user API.

Since the purpose of the case study is to evaluate a *gradually* typed system, the addition of type annotations to modules was not exhaustive. Instead, the key modules of each program were ported, with an emphasis on modules that used objects and classes.

3.2 The Process

Some of the code comes with comments, behavioral contracts, or documentation that describe the "types" of methods and fields. Injecting types into such pieces of code is often straightforward, though the specifications are sometimes out-of-sync with the program logic. When the code lacks specifications, the maintainer must reconstruct them from the program logic. This ranges from easy (e.g., for fields with an initial value) to difficult (e.g., methods with complicated invocation protocols).

Over the course of the typing process, a developer iteratively acquires an understanding of the code and adds type annotations until the type-checker is satisfied. In practice, the developer may need to modify the program logic or add assertions or casts to force type checking. Even after the type-checker approves the component, the typing effort is not over. Components that interact with other untyped components do not run correctly if an impedance mismatch exists between the types specified in an import statement and the run-time behavior of the untyped components. Hence, the developer runs the program on its test suite and improves the types in response.

3.3 Quantitative Results

Concerning metrics, we follow the precedent for functional Typed Racket [47]. These metrics are chosen to judge whether a developer can gradually equip a code base with types: the size of the code plus the number of type declarations, type annotations, and type assertions. The latter are important because the annotations are also software artifacts for which a developer must accept maintenance responsibility. In addition, we report how many changes

Program	Let	TT	BB	Esq	Mi	Card	Mdn	DB	Acq	GF	Ace	Sld
Lines	216	218	1077	177	533	620	328	749	1419	443	333	357
% Increase	2	7	11	11	13	19	16	23	20	11	5	15
Useful ann.	11	9	85	8	22	38	27	31	83	30	13	21
λ: ann.	0	0	15	4	38	5	4	3	19	12	10	2
Other ann.	14	7	29	0	4	17	0	2	12	5	0	4
Type def.	0	0	7	0	6	5	1	13	21	1	2	2
Typed req.	0	0	20	0	0	1	3	35	71	3	2	0
Assert/cast	4	3	25	1	10	4	11	5	13	3	0	9
Ann./100L	12	8	12	7	13	10	10	5	9	11	8	8
Problems	4	3	12	1	5	5	2	6	4	1	2	1
Fixes	1	1	1	0	1	1	2	2	0	1	0	1
Theo. princ.	4	1–4	1–4	–	1,3	1,3,4	2–4	1–4	1,3	–	–	–
Prac. princ.	D	A,D	A,B D	D	A,B D	A,B D	B,D	A–D	A–E	A	A	A,B D
Time	–	–	–	–	9h	7h	7h	7h	11h	1h	4h	5h
Difficulty	⋆	⋆	⋆⋆⋆	⋆	⋆	⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆	⋆	⋆

Figure 7 Case study results.

to the program logic are needed to accommodate the type system, because such changes may potentially alter the program's behavior.

Figure 7 reports the results in a table. The detailed interpretations of the rows are as follows. The *Lines* row indicates the total number of lines in the ported program while the *% Increase* row denotes the percentage of the total added by porting. The *Useful annotations* row consists of the number of identifiers given types; for example, the method type in the following excerpt from Big-Bang counts as useful:

```
(: show : Image -> Void)
(define/public (show pict0) ...)
```

In contrast, annotations added for the sake of the type-checker are not counted in this category. The *λ: annotations* row contains the number of annotations for function parameters of typed lambda expressions; these do not count toward the useful category because their types are often obvious from context, but the type system cannot infer them. The remaining *Other annotations* category covers the rest.

The *Type definition* row describes the number of type definitions added. As section 2 explains, many uses of `define-type` introduce names for class types. The *Typed require* row counts the number of bindings that are imported from untyped code using `require/typed`. The *Assertions / casts* row counts the number of assertions and casts used to assure the type-checker. The *Ann. / 100L* row shows the number of type annotations per one-hundred lines of code, rounded to the nearest integer.

The *Fixes* row indicates the number of error (correction)s due to types while *Problems* counts the changes made to circumvent limitations or over-approximations in the type-checker.

The *Theoretical/Practical principles* rows note principles (1-4) and (A-E) from figure 2 that apply to the code base. The *Time taken* row measures the number of hours (rounded up) taken to annotate and modify the code and finally the *Difficulty* describes the subjective difficulty of porting the code from ⋆ (easy) to ⋆⋆⋆ (hard).

All case studies rely on types for Racket's standard libraries, i.e., the base type environment. In addition to the core bindings provided by functional Typed Racket, our case studies use extra standard libraries such as the GUI libraries, drawing libraries, and core documentation

libraries. We do not count the annotations in the base type environment for the line numbers above because they are shared across all programs.

3.4 Qualitative Results

Principles. Figure 7 lays out which design principles from figure 2 were necessary for porting each code base. While subjective, we judged each code base with consistent criteria for each principle. For example, we decided that a code base required principle (1) if it used both class and object values. If a code base additionally used `Instance` type constructors, it fit principle (A). Principles (2) and (3) applied to any code base that used row polymorphism. Additionally, we checked (2) for any code base that used obviously structural types.

For (B) and (C), we determined whether the code base *directly* used `define-type` with a `#:implements` clause or with mutual-recursion, respectively. All of the case studies except Markdown, DB, and Acquire used mutually-recursive type definitions because of their dependence on the GUI standard library, but we did not count these indirect uses. For (4), we included any code bases that exported classes and/or objects to untyped code or imported them from untyped code. We recorded (D) if classes in the code base left out type annotations that TR would reconstruct. Only Acquire needed (E) due to exponential proxy layering.

Difficulty. The projects marked with ⋆⋆⋆ in the case studies share some of the following characteristics: (1) the data definitions and the code structure pose comprehension problems, (2) the code base uses language constructs that are difficult to describe with types, or (3) the control flow of the program makes the synthesis of type annotations difficult. The Markdown program fits the first case due to the use of 29 nested and recursive data structures in its logic. The Big-Bang program exemplifies the second characteristic. In Big-Bang, the primary class uses methods that act as mixins on other class values. Furthermore, the program also uses synchronization constructs for concurrency, syntactic extensions that construct methods, and I/O through the graphics layer and the networking library. Finally, the DB library uses complicated error handling that requires the programmer to track control flow when adding type annotations.

3.5 Problems and Fixes

The case studies identify several pain points in the system. Broadly speaking, these points take the form of syntactic overhead in type annotations or additional code necessitated by the type-checker. Here we list the three worst problems and explain how TR addresses them.

First, the `#:implements` shorthand for writing class types does not copy the types for constructor argument types because the constructor arguments may change. That is, a subclass does not necessarily use a superset of its parent constructor arguments. This design decision incurs some cost – in the form of larger types – for the case studies. To eliminate this limitation, TR now comes with a `#:implements/inits` form, which propagates the constructor types.

Second, the lack of occurrence typing for public fields forces a workaround to satisfy the type-checker. The workaround declares a local variable that holds the current value of the public field, which enables type refinement via occurrence typing on the local version. This works only when the field is not modified concurrently.

Third, the type system cannot propagate occurrence typing information or reason with enough precision about the expansion of syntactic extensions. These cases require re-writes with different functions or syntactic forms. An example of the former occurs in the Big-Bang

library, for which the excerpt `(inexact->exact (floor x))` is rewritten to `(exact-floor x)` to accommodate the type-checker. For the latter, our port of DB modifies a use of the `case` form, which provides simple pattern matching, to use `cond`, a general conditional form. Fortunately, both rewrites are simple and local.

3.6 Discussion

For the first hypothesis, we consider whether the effort of adding types to the code bases is reasonable. The overall increase in the number of lines for our object-oriented programs – about 15% across all code bases – is greater than the 7% increase across all of the programs ported in the functional world [47]. We conjecture two explanations: (1) structural type specifications for object-oriented programs are often larger and more complex than function types, and (2) the object-oriented part of Typed Racket lacks syntactic support for formulating concise types. In particular, Typed Racket does not derive class types from class definitions.

Additionally, the code bases that we could *not* include in our investigation are relevant for the first hypothesis. We rejected several code bases from the case study for three reasons: they use Racket's *first-class* modules, they call for the reflection API on records, or they require row polymorphism for objects. Only the third omission points to a flaw in our type system design; the first two are general Typed Racket limitations. The third point prevented the porting of only a single small module, and we thus consider the first hypothesis validated.

As for the second hypothesis, seven out of the twelve case studies require half or more of the theoretical design principles from figure 2. The ones that require the fewest are Esquire, Go Fish, Aces, and Slidey, which are all self-contained and do not use mixins. Due to their self-contained nature, these programs also do not require extensive contract checking.

For the third hypothesis, we also see that most of the case studies rely on three or more of the practical design principles. The least directly used is mutually-recursive type definitions (C). As noted above, however, the feature is heavily employed in the common base type environment and is therefore indirectly used everywhere.

4 Performance Evaluation

Gradual typing suggests that a programmer who performs maintenance work on existing code (re)discovers the types behind the design and adds them to the code base. Each conversion may put up a new higher-order contract boundary between two components, which may have negative consequences for the performance of the overall system. In theory, a completely converted program – without casts or type assertions – should have the same performance as a completely untyped one, because the type checker simply removes types in this case.[7]

A performance evaluation must therefore consider all possible paths from an untyped system to a completely typed one through the lattice of possibilities. Thus far, the gradual typing literature has not presented any results for such experiments. In this section we present the first results of such a performance analysis and explain its role as a formative element of our design process. Specifically, we explain our methodology in some detail, present the results of two experiments, and explain the performance pitfalls that these formative evaluations found and our fixes.

[7] Typed Racket currently performs local optimization on some simple datatypes [50].

4.1 Methodology

The runtime cost of a (sound) gradual type system is a function of boundaries between typed and untyped pieces of code. As the programmer chooses to create such boundaries via type annotations, the system migrates through a space of possible boundary configurations. In Typed Racket, these boundary configurations are determined mostly by which modules are typed and untyped. The path through the space of mixes of typed and untyped modules starts at the fully untyped program. At each step, one more piece is annotated with types.

For example, the lattice in figure 8 shows the simplified configuration space (of four modules) for Acquire. Each node in the lattice contains four boxes whose shading indicates whether it is typed or untyped. The horizontal bottom box represents an I/O library that remains untyped throughout the process.[8]

During the implementation of Typed Racket, we used the following, modest working hypothesis based on Safe TypeScript's experience of 72x slowdowns [32, p. 12]:

> *no path through the lattice of configurations imposes an order-of-magnitude degradation of run-time performance at any stage.*

Pragmatically put, a programmer may add types anywhere in the program without slowing down the program more than a factor of 10.

We used two of our case studies to investigate this hypothesis: Acquire and Go-Fish. For each case study, we converted the primary modules – excluding infrastructure modules such as the I/O library mentioned above – and then scripted the generation of all possible configurations.[9] Each configuration was run 30 times on a stand-alone Linux 3.16 computer with an Intel Core i7-3770K CPU and 32GB of memory.

4.2 Results and Preliminary Interpretation

Figure 8 annotates the lattice of configurations with timings – shown below the modules – that are normalized to the baseline of the fully untyped configuration. The figure also displays normalized standard deviations. An inspection of the lattice reveals that choosing the outermost paths degrades the performance quickly, while some of the innermost paths reduce the performance in a gradual manner. At the worst points, a gradually typed program is almost 40% slower than an untyped program; at the top, we find a fully typed program that is still 39% slower than the fully untyped program.[10]

A large fraction of the additional cost is due to the boundary between the untyped library modules in Acquire (the horizontal white box in the figure) and the typed modules. Some of the cost is due to the boundaries among the four central modules. Critically no configuration slows down performance by an order of magnitude.

Figure 9 collects the results of evaluating the lattice for the four modules[11] in Go-Fish. Many paths from bottom to top go through only one point in which the performance

[8] Adding coarse types to this library is easy but useless; adding precise types is currently impossible.

[9] The scripting required introducing additional typed wrapper modules in some cases to provide extra type signatures. We consider these wrappers to be part of the infrastructure modules.

[10] Earlier versions of Typed Racket caused overheads of up to around 3.4x slowdown on Acquire but still did not exceed 10x. The overhead was brought down to current levels by compiling object types to more efficient contract forms.

[11] We started out with five modules in the Go-Fish experiment but found that one module was never run – and thus did not affect the benchmark runtime – therefore we chose to keep it untyped to make the results clearer.

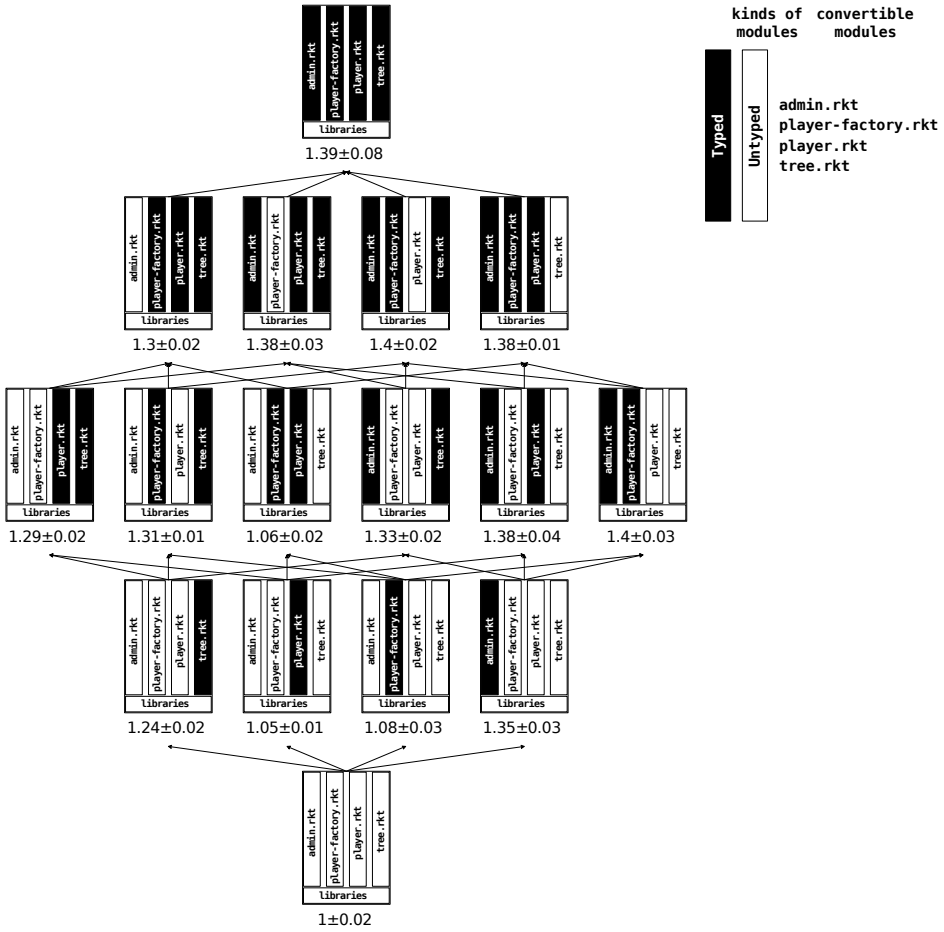

Figure 8 Lattice results for Acquire.

degradation is worse than 10%. There is also a single path along which the overhead never exceeds 5%. In no case does the timing exceed 1.33x of the baseline. The lattice also shows which modules add costs when they are typed. For example, the lattice shows that the typed `classes.rkt` module adds significant overhead unless `gofish.rkt` is also typed.

4.3 Problems and Fixes

The first implementation of OO support for Typed Racket falsifies even the modest working hypothesis, revealing a major problem in the language's run-time infrastructure. Concretely, running some configurations exhibits exponential degradation in performance, both in terms of space and time consumption. Indeed, some configurations no longer terminate. This section explains the problem and sketches how our second implementation of OO support for Typed Racket solves it. This solution is an implementation of principle (E) from figure 2.

As mentioned in section 2.1, Typed Racket compiles types for module exports and imports into Racket's contracts so that untyped modules cannot violate the type invariants. For plain values, a contract just checks that the value conforms to the type. For higher-order values, e.g., objects and classes, the contract system wraps the value with a proxy value – called

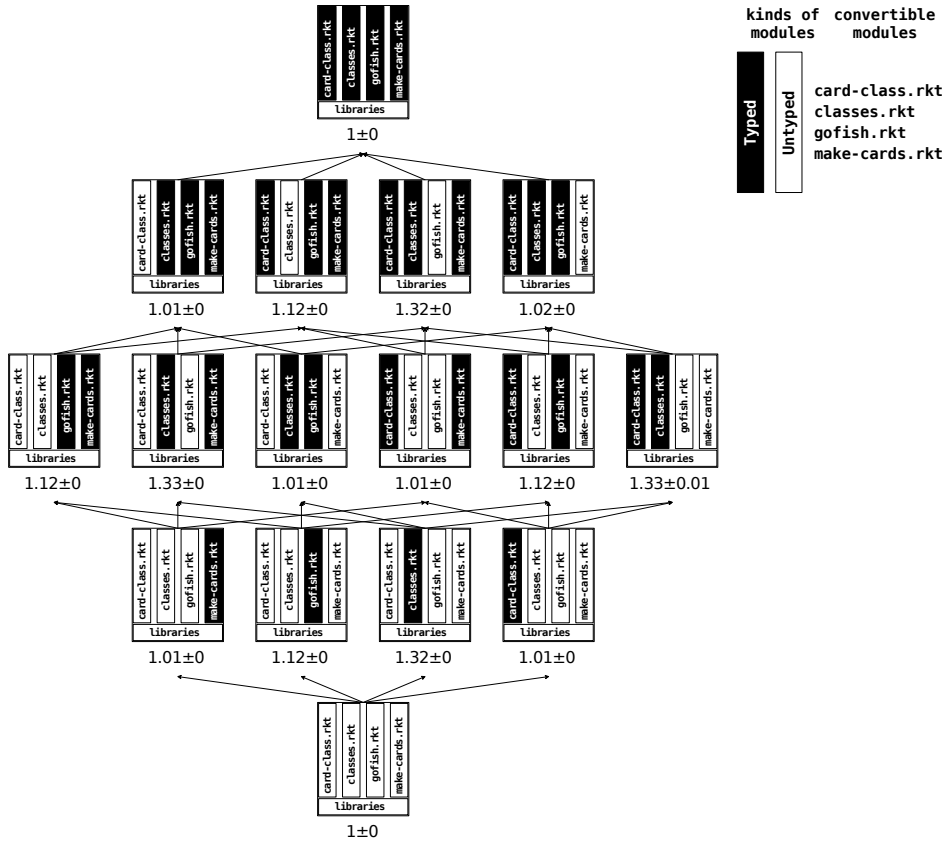

Figure 9 Lattice results for Go-Fish.

chaperones [45] in Racket – that interposes on all accesses and method calls. If the proxy wrapper discovers any bad behavior, it blames the module that agreed to the contract and violated it, which is critical because wrapped values can flow arbitrarily far away from the contract boundary before a violation shows up.

Given this background, the problem is easily explained. Every time a higher-order value flows from a typed module to an untyped one or vice versa, the run-time system wraps it with a proxy layer. A method call may add another layer because the method's signature may impose type invariants. If this boundary-crossing occurs in a loop, layers can quickly accumulate. The growth in layers is exponential with respect to the number of boundary crossings. In the worst case, each round-trip corresponds to a doubling in the number of layers, which explains the exponential consumption of space (wrappers) and time (interposition).

Figure 10 illustrates the idea with a minimal example using a class with a single method; the example distills a problematic two-module fragment from Acquire. Method `m` in `obj` is defined and exported (to `"B.rkt"`) with a recursive, higher-order contract – a common phenomenon in the object-oriented world where classes and objects refer to themselves. The expression `(loop obj)` kicks off the program, starting a loop that calls the method `m` on the object. While this loop uses a finite amount of space in the absence of contracts, the evaluation of `(send obj m obj)` applies the domain contract on `m` to the argument, which is `obj` including its sole method.

```
#lang racket ; A.rkt
(provide (contract-out [obj bubble/c]))

(define bubble/c
  (recursive-contract
   (object/c [m (->m bubble/c bubble/c)])))

(define obj
  (new (class object%
         (super-new)
         (define/public (m x) x))))
```

```
#lang racket ; B.rkt

; The driver module, kicks off the
; problematic loop below

(require "A.rkt")

(define (loop obj)
  (loop (send obj m obj)))

(loop obj)
```

Figure 10 Exponential wrapping.

When the loop starts up, `obj` is already wrapped in one proxy layer due to `bubble/c` from `contract-out`. The domain contract of `m` wraps another layer around `obj`, which increases the number of chaperones to two. Once `m` returns `obj`, the range contract on `m` wraps a final layer around the object, increasing the number to three. For the next iteration, `obj` starts with three layers that enforce `bubble/c`. Since `obj` is *both* the target and the argument in `(send obj m obj)`, each of the domain and range contracts are applied three times, once per existing layer. In short, the number of layers doubles to six. This doubling occurs on each iteration and thus causes exponential memory use.

Our revised implementation of TR solves this problem, loosely based on two theoretical investigations mentioned above. Concretely, Herman et al. [25] compile the contracts in their surface language to a Henglein-style coercion calculus, in which multiple levels of coercions can be eliminated – though without respect for blame information. Siek and Wadler [41] collapse layers of coercions into a representation that includes the greatest lower bound of all types involved in a sequence of wrappers – and thus preserve blame information. Although these theoretical solutions do not apply to Racket's contract system directly, the idea of collapsing layers is applicable to our system.

The current implementation of TR improves the first one in two regards. First, Racket's revised contract system checks whether an existing contract implies one that is about to be wrapped; if so and if the blame information is identical, the new wrapper is not created. Second, Racket's revised proxy mechanism allows dropping a layer in special cases. In particular, some proxies can be marked as containing only metadata with no interposition functions. The run-time system allows such proxy layers to be removed and replaced. By encoding blame information in these metadata proxies, it is possible to replace layers instead of adding redundant ones. This optimization currently works only for object contracts; we intend to generalize it for other language constructs in the future. Together these two changes removed all performance obstacles and enable the revised TR implementation to validate our modest performance hypothesis.

4.4 Threats to Validity

Our formative performance evaluation suffers from some shortcomings that suggest it might not be representative for a summative evaluation. First, the experiments evaluate only the overhead of the contracts created by the typed-untyped boundary. Second, they ignore the overhead due to modifications of the program logic. If a programmer changes the code to

accommodate the type checker or inserts a cast, the current evaluation attributes this cost to the typed/untyped boundaries.[12] Third, our module boundaries may not be representative because we merged some smaller modules (e.g., 10 lines of code) in the code bases into larger ones in order to reduce the lattice size. Since the lattice grows exponentially in the number of modules, exploring the full lattice would take too much time. Fourth, the top of the lattice does not correspond to a program in which every single module is exhaustively typed; infrastructure modules and difficult-to-type modules are left out. Finally, the experiments suffer from somewhat imprecise measurements. In particular, they execute untyped module in a `typed/no-check` mode, meaning the modules still load Typed Racket's run-time library.

5 Related Work

Since this paper reports on the transition from theoretical calculi [4, 40, 46] to full-fledged, gradually typed object-oriented programming languages, this section focuses on implementation efforts of gradual type systems and on prior implementations of typed languages with flexible class composition.

5.1 Gradualtalk

Typed Racket differs from Gradualtalk [3], a gradually typed dialect of Smalltalk, in two major ways. First, TR implements *macro*-level gradual typing using higher-order contracts as the enforcement mechanism at module boundaries. Meanwhile, Gradualtalk uses the *micro*-level approach pioneered by Siek and Taha [39], meaning that Gradualtalk programmers can freely omit type annotations. When they do, Gradualtalk injects the value into the `Dyn` type and downcasts it from there later.

Second, Gradualtalk does not require row polymorphism because Smalltalk projects rarely use dynamic inheritance with mixins or similar features. Classes are declared statically. In contrast, TR necessarily places more emphasis on structural types and row polymorphism to support the numerous dynamic uses of inheritance in Racket.

Due to the differences in the fundamentals, Typed Racket and Gradualtalk's evaluations are necessarily dissimilar. The Gradualtalk evaluation consists of porting an impressive corpus of nearly 19k lines, with the largest typed component consisting of over 9k lines. These Gradualtalk components make significant use of the `Dyn` type, which we conjecture makes porting large numbers of lines easier than in Typed Racket, likely trading type precision. More precisely, for every difficult-to-type phrase, a programmer can use `Dyn` and avoid the hard work of developing a precise type; conversely, replacing uses of `Dyn` may trigger non-local program changes. Qualitatively, the difference in type precision manifests itself at run-time. With `Dyn` types, the dynamic portions may be deeply intertwined with typed portions and thus many more code paths may emit a coercion failure.

In addition, the flavor of ported components differs. Gradualtalk's evaluation includes the Kernel project, which contains the core classes of Smalltalk. Racket's use of classes in the core is limited to those few places where extensibility or GUI hierarchies are needed. Our case studies therefore focus on GUI programs or those, such as Markdown or DB, which are built for extensibility.

[12] The `Go-Fish` experiment runs in headless mode because casts in the GUI code are excessively expensive at the moment. We are investigating the cause.

Concerning performance, the Gradualtalk evaluation does not consider the porting process as a whole. Allende et al. [4] do evaluate the performance of several cast insertion strategies using microbenchmarks.

5.2 Reticulated Python

Like Gradualtalk, Reticulated Python [52] (henceforth Reticulated) implements micro-level gradual typing with `Dyn` types. In an attempt to overcome performance problems, Reticulated implements three styles of cast semantics with different design tradeoffs. The *guarded* semantics is most similar to Typed Racket's use of proxy objects to implement higher-order casts. Unlike the latter, Reticulated uses "threesomes" to avoid repeated proxying. TR does not use "threesomes" because Racket's underlying contract language is more expressive than Reticulated's cast language. Furthermore, the runtime support for contracts (i.e., chaperones) enforces more stringent restrictions than Reticulated's proxies. Like TR, the Reticulated evaluation found that object identity posed a challenge for porting programs in the guarded semantics. The *monotonic* semantics [38] avoids proxying while maintaining blame, at the cost of potential extra errors when interacting with untyped code, but has not yet been fully evaluated in Reticulated.

For recursive type aliases, Reticulated uses a fixpoint computation over its class declarations to determine the recursive object types to substitute into class bodies [52, § 2.1.3]. Meanwhile, TR's `define-type` allows the encoding of general mutual recursion between *any* type declarations.

Reticulated's mostly qualitative evaluation does not analyze performance concerns.

5.3 Thorn

Instead of gradually layering typed reasoning on an untyped language, a designer may also choose to embed design elements of untyped languages in a statically-typed language. Notably, the Thorn language takes this approach to support flexible object-oriented programming via *like* types [7, 55]. The uses of a variable annotated with a like type are statically checked, but at run-time any value may flow into such variables. While Thorn's design goals include providing the "flexibility of dynamic languages," its design explicitly leaves out the "most dynamic features" such as dynamic class composition [55]. In contrast, we aim TR specifically at augmenting existing code bases, and thus it necessarily supports the dynamic features that are in use.

In addition, the goals of Typed Racket and Thorn differ in their treatment of blame and when run-time errors may occur. In TR, most run-time checks occur at module boundaries and thus most mismatches are signaled when a module is imported. Thus, if an untyped object imported with a type is missing any specified methods, the contract system immediately blames the untyped module. Thorn, on the other hand, checks method presence only when the object flows into an expression in which the method is used. Thus, a tradeoff is made between flexibility and immediate checking of specifications. Furthermore, Thorn provides no equivalent of blame tracking, trading precision of debugging information for performance.

5.4 Typescript and Hack

Industrial designers of programming languages have started to adopt ideas from the gradual typing research community. In particular, both Typescript and Hack allow programmers to add types to programs in their respective base languages, JavaScript and PHP. These efforts,

like Typed Racket, focus on supporting the idioms in the underlying languages such as traits – an alternative to mixins that emphasize horizontal composition – or prototypes. They make no effort, however, to put the interoperation of typed and untyped code on a sound footing.

Recently, Rastogi et al. [32] proposed Safe TypeScript, which enables safe interoperation for TypeScript. Their approach differs from Typed Racket in using run-time type information for casts whereas TR erases all types after compiling to contracts. Their performance evaluation measures the performance overhead of casts on several Octane benchmarks in two modes, with and without type annotations. Superficially, this is similar to testing TR in both the fully untyped (bottom) and fully typed (top) modes. However, they are not directly comparable because Safe TypeScript incurs a heavy (up to 72x) overhead with no type annotations while untyped Racket code does not incur any overhead until it interacts with a typed module.

5.5 Soft and Strong Type Systems for Dynamic Languages

Type systems that accommodate reasoning for untyped programs have been proposed for many languages. Early work includes soft typing for Scheme [9, 13, 31, 54], polymorphic type inference for Scheme [24] based on the *dynamic typing* [23] formalism, the Strongtalk project [8] for statically-typed Smalltalk with mixins, and Marlow and Wadler [28]'s work on a type system for Erlang. These early proposals do not support interoperation as defined by gradual typing. In soft-typing, run-time checks are inserted where the type system cannot reason with the given rules. The checks come without blame. Strongtalk provides an idiomatic type system for Smalltalk, but offers only "downward compatibility" [8] (i.e., Strongtalk code elaborates to valid Smalltalk). The elaboration is not sound for interoperation with Smalltalk in the sense of gradual typing. Several ideas used in Strongtalk, e.g., "brands" and "protocols", are relevant for future extensions to Typed Racket such as nominal typing and more concise types.

More recently, several designs in this space use a variety of techniques such as type inference, dependent types, and refinement types to support idioms in dynamically-typed programs. DRuby [20] uses type inference to discover types errors in Ruby programs and inserts run-time checks if the programmer supplies type annotations. Dependent JavaScript [10] supports JavaScript idioms found in real world programs through the use of dependent types with a refinement logic, off-loading some of the reasoning to an SMT solver. While these systems are not gradually-typed, their techniques will be helpful for future improvements to gradual typing of objects.

5.6 Types for Mixins and First-class Classes

Our work is inspired by a long line of research on semantics and type systems for mixins and objects. The literature on mixins has focused on class-based languages, many inspired by Java or Smalltalk. In the object world, classes are encoded as syntactic sugar as in the σ-calculus [1] or ML-ART [34].

Many models of mixins or first-class classes have been proposed for Java-like languages: Flatt et al. [17]'s MixinJava, Ancona et al. [6]'s Jam, McDirmid et al. [30]'s Jiazzi, Allen et al. [2]'s MixGen, Kamina and Tamai [27]'s McJava, and Servetto and Zucca [37]'s MetaFJig. Other designs instead provide *traits* [35, 42], which emphasize non-linear composition using rich operations on trait members. OCaml's addition of first-class modules [19] enables a kind of run-time class composition as well. These designs all provide flexible class composition, but typically do not provide the ability to compose classes at run-time.

6 Lessons Learned and Future Work

This paper explains what it takes to turn a theoretical calculus of gradual typing into a full-fledged object-oriented language that respects pre-existing constraints, especially dynamic class composition idioms. The key insights are the theoretical and practical design principles that are applicable across the board. In addition, the paper introduces a novel performance evaluation framework for gradual typing. While the performance results are restricted to the formative part of our design work, they have confirmed a long-held belief among researchers in the community; no other gradual typing project has reported anything comparable. The current implementation of TR owes its shape to negative results from this evaluation.

Our work suggests two kinds of future efforts. First, we need to scale up the formative performance evaluation to a summative one that uses a variety of programs. We also intend to use the framework on a different gradually typed language, e.g., Reticulated Python, that takes a micro-level approach to gradual typing. Doing so will confirm that this approach is useful across the board. Second, the performance framework also suggests that programmers need tailored performance-measuring tools that help them find a path from slow-performing configurations to better ones. For Typed Racket, we intend to investigate the use of profiling techniques [43] that pinpoint the most expensive boundaries so that programmers can eliminate those first.

Acknowledgments. The authors wish to thank Leif Andersen, Ben Greenman, and Vincent St-Amour for their comments on early drafts and for discussions about the research itself. We also thank the anonymous reviewers for their feedback.

The work was partially supported by a DARPA grant at Northeastern and Utah, an NSA grant at Indiana, and several NSF grants at all four sites.

References

1 M. Abadi and L. Cardelli. A Theory of Objects. Springer-Verlag, 1996.
2 E. Allen, J. Bannet, and R. Cartwright. A First-class Approach to Genericity. In *Proc. OOPSLA*, pp. 96–114, 2003.
3 E. Allende, O. Callaú, J. Fabry, É. Tanter, and M. Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, 2013.
4 E. Allende, J. Fabry, and É. Tanter. Cast Insertion Strategies for Gradually-Typed Objects. In *Proc. DLS*, pp. 27–36, 2013.
5 J. An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic Inference of Static Types for Ruby. In *Proc. POPL*, pp. 459–472, 2011.
6 D. Ancona, G. Lagorio, and E. Zucca. Jam – A Smooth Extension of Java with Mixins. In *Proc. ESOP*, pp. 154–178, 2000.
7 B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn: Robust, Concurrent, Extensible Scripting on the JVM. In *Proc. OOPSLA*, pp. 117–136, 2009.
8 G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proc. OOPSLA*, pp. 215–230, 1993.
9 R. Cartwright and M. Fagan. Soft Typing. In *Proc. PLDI*, pp. 278–292, 1991.
10 R. Chugh, D. Herman, and R. Jhala. Dependent Types for JavaScript. In *Proc. OOPSLA*, pp. 587–606, 2012.
11 R. L. Constable and N. P. Mendler. Recursive Definitions in Type Theory. Cornell University, TR 85-659, 1985.

12 M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. A Functional I/O System (or Fun for Freshman Kids). In *Proc. ICFP*, pp. 47–58, 2009.

13 C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *Proc. PLDI*, pp. 23–32, 1996.

14 M. Flatt, E. Barzilay, and R. B. Findler. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *Proc. ICFP*, pp. 109–120, 2009.

15 M. Flatt, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Programming Languages as Operating Systems (or Revenge of the Son of the Lisp Machine). In *Proc. ICFP*, pp. 138–147, 1999.

16 M. Flatt, R. B. Findler, and M. Felleisen. Scheme with Classes, Mixins, and Traits. In *Proc. APLAS*, pp. 270–289, 2006.

17 M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *Proc. POPL*, pp. 171–183, 1998.

18 M. Flatt and PLT. Reference: Racket. PLT Inc., PLT-TR-2010-1, 2010. `http://racket-lang.org/tr1/`

19 A. Frisch and J. Garrique. First-class Modules and Composable Signatures in Objective Caml 3.12. In *Proc. ML Workshop*, 2010.

20 M. Furr, J. An, J. S. Foster, and M. Hicks. Static Type Inference for Ruby. In *Proc. SAC*, pp. 1859–1866, 2009.

21 M. Greenberg. Space-Efficient Manifest Contracts. In *Proc. POPL*, pp. 181–194, 2015.

22 S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik. An Empirical Study on the Impact of Static Typing on Software Maintainability. *Empirical Software Engineering*, pp. 1–48, 2013.

23 F. Henglein. Dynamic Typing: Syntax and Proof Theory. *Science of Computer Programming* 22(3), pp. 197–230, 1994.

24 F. Henglein and J. Rehof. Safe Polymorphic Type Inference for a Dynamically Typed Language: Translating Scheme to ML. In *Proc. FPCA*, pp. 192–203, 1995.

25 D. Herman, A. Tomb, and C. Flanagan. Space-efficient Gradual Typing. *HOSC* 23(2), pp. 167–189, 2010.

26 L. Ina and A. Igarashi. Gradual Typing for Generics. In *Proc. OOPSLA*, pp. 609–624, 2011.

27 T. Kamina and T. Tamai. McJava – A Design and Implementation of Java with Mixin-Types. In *Proc. APLAS*, pp. 398–414, 2004.

28 S. Marlow and P. Wadler. A Practical Subtyping System for Erlang. In *Proc. ICFP*, pp. 136–149, 1997.

29 J. Matthews and R. B. Findler. Operational Semantics for Multi-Language Programs. *TOPLAS* 31(3), pp. 12:1–12:44, 2009.

30 S. McDirmid, M. Flatt, and W. C. Hsleh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proc. OOPSLA*, pp. 211–222, 2001.

31 P. Meunier, R. B. Findler, and M. Felleisen. Modular Set-Based Analysis from Contracts. In *Proc. POPL*, pp. 218–231, 2006.

32 A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & Efficient Gradual Typing for TypeScript. In *Proc. POPL*, pp. 167–180, 2015.

33 B. M. Ren, J. Toman, T. S. Strickland, and J. S. Foster. The Ruby Type Checker. In *Proc. SAC*, pp. 1565–1572, 2013.

34 D. Rémy. Programming Objects with ML-ART an Extension to ML with Abstract and Record Types. In *Proc. TACS*, pp. 321–346, 1994.

35 N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable Units of Behaviour. In *Proc. ECOOP*, pp. 248–274, 2003.

36 M. Scriven. The Methodology of Evaluation. Perspectives of Curriculum Evaluation. Rand McNally, 1967.

37 M. Servetto and E. Zucca. MetaFJig: a Meta-circular Composition Language for Java-like Classes. In *Proc. OOPSLA*, pp. 464–483, 2010.

38 J. G. Siek, M. M. Vitousek, M. Cimmini, S. Tobin-Hochstadt, and R. Garcia. Monotonic References for Efficient Gradual Typing. In *Proc. ESOP*, pp. 432–456, 2015.

39 J. G. Siek and W. Taha. Gradual Typing for Functional Languages. In *Proc. SFP*, 2006.

40 J. G. Siek and W. Taha. Gradual Typing for Objects. In *Proc. ECOOP*, pp. 2–27, 2007.

41 J. G. Siek and P. Wadler. Threesomes, with and without blame. In *Proc. POPL*, pp. 365–376, 2010.

42 C. Smith and S. Drossopoulou. Chai: Traits for Java-Like Languages. In *Proc. ECOOP*, pp. 453–478, 2005.

43 V. St-Amour, L. Andersen, and M. Felleisen. Feature-specific profiling. In *Proc. CC*, pp. 49–68, 2015.

44 T. S. Strickland and M. Felleisen. Contracts for First-Class Classes. In *Proc. DLS*, pp. 97–112, 2010.

45 T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and Impersonators: Run-time Support for Reasonable Interposition. In *Proc. OOPSLA*, pp. 943–962, 2012.

46 A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Gradual Typing for First-Class Classes. In *Proc. OOPSLA*, pp. 793–810, 2012.

47 S. Tobin-Hochstadt. Typed Scheme: From Scripts to Programs. Ph.D. dissertation, Northeastern University, 2010.

48 S. Tobin-Hochstadt and M. Felleisen. Interlanguage Migration: from Scripts to Programs. In *Proc. DLS*, pp. 964–974, 2006.

49 S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. In *Proc. POPL*, pp. 395–406, 2008.

50 S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as Libraries. In *Proc. PLDI*, pp. 132–141, 2011.

51 Typescript Language Specification. Microsoft, Version 0.9.1, 2013.

52 M. M. Vitousek, A. Kent, J. G. Siek, and J. Baker. Design and Evaluation of Gradual Typing for Python. In *Proc. DLS*, pp. 45–56, 2014.

53 R. Wolff, R. Garcia, É. Tanter, and J. Aldritch. Gradual Typestate. In *Proc. ECOOP*, pp. 459–483, 2011.

54 A. K. Wright and R. Cartwright. A Practical Soft Type System for Scheme. *TOPLAS* 19(1), pp. 87–152, 1997.

55 T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating Typed and Untyped Code in a Scripting Language. In *Proc. POPL*, pp. 377–388, 2010.