Migratory Typing: Ten Years Later*

Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, Asumu Takikawa[1]

1 **PLT** *@racket-lang.org

Abstract

In this day and age, many developers work on large, untyped code repositories. Even if they are the creators of the code, they notice that they have to figure out the equivalent of method signatures every time they work on old code. This step is time consuming and error prone.

Ten years ago, the two lead authors outlined a linguistic solution to this problem. Specifically they proposed the creation of typed twins for untyped programming languages so that developers could migrate scripts from the untyped world to a typed one in an incremental manner. Their programmatic paper also spelled out three guiding design principles concerning the acceptance of grown idioms, the soundness of mixed-typed programs, and the units of migration.

This paper revisits this idea of a migratory type system as implemented for Racket. It explains how the design principles have been used to produce the Typed Racket twin and presents an assessment of the project's status, highlighting successes and failures.

1998 ACM Subject Classification D.3 Language Design, F.3.3 Type Structure

Keywords and phrases design principles, type systems, gradual typing

Digital Object Identifier 10.4230/LIPIcs...0

1 Migratory Typing

In the 1970s and 80s, developers chose Lisp for its flexibility, its libraries, and occasionally Lisp-specific hardware. They argued that higher-order functions, class systems, mixin classes, automatic memory management and the absence of *static* types enhanced their productivity. To their credit, statically typed languages with similar features did not exist, and existing compilers used types mostly as hints for choosing data representations and optimizations.

In the 1990s and 00s, developers continued to opt into a wide array of scripting languages that, like Lisp, lacked static types and simultaneously supported a rich collection of powerful features. System administrators picked Perl as their scripting language. Python and Ruby (on Rails) emerged as the primary vehicles for web-server extensions. PHP and JavaScript became the assembly languages of the web (browser). Java, the first soundly typed production language, tried and failed to take the place of these languages.

When the senior authors launched their pedagogical outreach project [15, 16], they settled on an un(i)typed[1] implementation language, Racket, for a mix of the same reasons that the Lispers and their successors had offered before them. They knew that this choice would enable them to rapidly build and deploy teaching languages [17, 18]. Within a few short

* Over 10 years, this work was partially supported by our host institutions (Chicago, Northeastern, Northwestern, Indiana, and Utah) as well as AFOSR, DARPA, Mozilla, NSA, and NSF.

[1] With this strange spelling, we emphasize that we are thinking of "safe" untyped languages, and we acknowledge that Dana Scott suggested "unityped" as an alternative characterization.

years, they then faced the task of maintaining a code repository of 500,000 lines of untyped Racket and an exponentially growing web repository of Racket libraries.

Regardless of why developers—consciously or not—choose to develop code in an untyped language, they sooner or later end up in a similar situation. While they design code with type-like ideas in mind, they cannot write down these insights within their programming language and get them cross-checked against code. Every time they have to modify or extend a component, their first chore is to reconstruct the type-like design ideas of the original creator. As any experienced developer knows, this task is time consuming and error prone.

This scenario clearly explains why the authors claim that

even in an untyped world, types are critical for the cost-effective maintenance of software.

Once reliable type information becomes available, developers can comprehend a code base more easily than without types. Similarly, IDEs can exploit types to assist developers with various tasks, because types also help IDEs to reason about code. When code is changed, type checking can catch basic errors even before the unit tests uncover those. Finally, compilers rely on type information to produce performant target code, and software developers may wish to squeeze extra performance percentages out of their untyped code base.

And now the question arises how to equip a large, untyped code base with useful types. The lead authors' vision paper [49] proposes a linguistic solution to this research problem:

> *the creation of an explicitly-statically typed twin of the given programming language to which developers can migrate pieces of untyped code in an incremental manner never losing the ability to test and deploy the software system.*

Their paper also spells out the three basic principles of what such a typed twin language should look like, with direct implications for its development:

1. The type system must cause as few perturbations to the code as possible so as not to "enbug" the code during migration (see section 3).
2. The mix of untyped and typed code must remain executable, and it must guarantee the soundness of type information, which calls for run-time enforcement [30] (see section 4).
3. The "unit of migration" must satisfy two opposing desires: (a) It must be small enough to encourage the incremental migration of untyped code into the typed twin language. (b) It must also be large enough to keep values from crossing the language boundary too often because every crossing may trigger a run-time check (see section 5).

Section 7 provides an assessment of the state of the research program and lessons for language designers. The next section re-tells the pre-history of types for untyped languages from the perspective of the second author, motivating the idea of a typed twin language.

2 Some Pre-history

Most people reply with "type inference" when confronted with the problem of figuring out the types of programs in untyped languages, a solution that comes with an early historical justification [44]. They know about ML, Haskell, and Hindley-Milner type inference, but they rarely appreciate how subtle type inference is. Roughly speaking, the type-inference problem asks for the restoration of missing type declarations for variables and functions—and it thus seems a perfect match for the stated problem. Mathematically speaking, the problem asks for the solution of a system of equations over an uninterpreted algebra of types whose variables stand in for the omitted type declarations:

$$\overrightarrow{\text{(variable expression over uninterpreted type algebra)} = \text{(type)}}$$

Depending on the precise formulation, however, the problem is easily reducible to the Halting Problem or similarly unsolvable problems; for an early example, see Boehm's 1985 paper [6] on the undecidability of a form of polymorphic type inference.

In the context of untyped programming languages, the problem takes on a different form. By definition, a program in an untyped language contains neither any type definitions (say, for recursive data types) nor any declarations (say, for functions). Instead, values come with tags that specify which operations are applicable. Hence the most straightforward way to reason about untyped code is to think in terms of sets and subsets of values, which means the natural mathematical problem statement involves systems of *in*equations, not equations:

$$\overrightarrow{(\text{variable expression over uninterpreted type record algebra}) \subseteq (\text{type})}$$

The challenge is to develop a method for solving such systems of inequations.

Over the past 30 years, researchers have essentially developed two solution methods:[2]

1. Inspired by operations research, Mike Fagan's "soft typing" approach [13, 11] uses so-called *slack variables* to turn a system of inequations into a system of equations:

   $$\overrightarrow{(\text{variable expression over uninterpreted type record algebra}) \oplus \text{slack-variable} = (\text{type})}$$

   At this point it is possible to use a relatively standard[3] version of the Hindley-Milner algorithm. If a solution assigns a non-empty (record) type to a slack variable, the program is not typable in the given algebra. The essence of soft typing is to turn a non-empty slack variable into a cast that camouflages the associated type error. Wright [56, 58]'s dissertation shows how to scale the idea to R4RS [12].
2. Inspired by static program analysis, Flanagan's "static debugging" approach [20, 22, 23] uses a modified transitive closure algorithm to solve the system of inequalities *directly*.

In the experience of the second author[4] of this paper, who used all of the above systems, plain static type inference does not work well for a code base in an untyped language—without modifying the code pervasively, which would explicitly contradict the idiomacy constraint. In a nutshell, any form of inference suffers from some of the following problems:[5]

syntactic brittleness Both approaches infer types from the syntactic shape of the program. A simple, semantics-preserving modification may turn a two-line type description into one of ten lines—to the great consternation of a practicing programmer.

non-actionable type errors Going back to 1987 [55], researchers have studied the problem of deciphering error messages in the Hindley-Milner setting and making them actionable. Not surprisingly, Wright's Soft Scheme exhibits the same problem as ordinary Hindley-Milner languages. Although people have made some progress on this problem in the past ten years, all of this work applies to the standard type algebra, and it remains an open question how (well) their solutions apply to inference in a completely untyped setting.

non-modularity Flanagan's approach greatly simplifies the search for type errors in untyped programs. Unlike unification, the central piece of the Hindley-Milner algorithm, his algorithm is directional and thus connects the source of a type error with its actual

---

[2] Henglein [26] embeds the untyped language into a statically typed language using injections to, and projections from, a universal data type. The solution of IBM's FL group [1, 2] is like the second bullet.

[3] Fagan's approach uses Remy's type algebra of extensible records [34] and is thus not completely standard.

[4] He supervised Flanagan's dissertation, jointly supervised Wright's, and was deeply involved with Fagan's.

[5] Cardelli [10] argues that similar problems show up in a statically typed language with Hindley-Milner inference. He therefore suggests to think of type inference as a mere IDE tool.

manifestation. It comes with the separate disadvantage of requiring a whole-program analysis. Meunier's dissertation [32, 33] overcomes this new problem, but it demands that programmers specify contracts, which is essentially a step toward adding explicit types.

It is this last insight that caused the second author to look for a different solution, the construction of an explicitly and statically typed twin of the given untyped language.

3 Grown Idioms Require Accommodating Type Systems

Ideally the migration of untyped code to a typed twin language adds type definitions and declarations but leaves the code itself alone. While this constraint may not be obvious at the level of a single function, it is indisputable when it comes to modules of hundreds of lines of code or systems consisting of tens of thousands of lines. If developers are forced to change the code while migrating, they may end up "enbugging" their programs. Conversely, the goal of preserving existing code has serious implications for the design of the type system.

In this section, we explore these implications, starting at the function level. Figure 1 displays an idiomatic Racket function. As the comment in line 1 says, the code deals with trees of integers, where a leaf is represented with `#false` and interior nodes are three-element `list`s. The `snap` function traverses the tree, adding `1` to even integers, pruning odd ones, and growing the tree at its leafs. As in any statically typed language, a developer organizes functions on such trees according to the data type definition. Thus, the body of `snap` consists of a two-pronged conditional, one per branch in the informal data type specification. The function recurs on the two branches of a node, which are extracted with `first` and `third`.

Listing 1 A Racket function

```
1 ;; Tree is either #false or [list Tree Integer Tree]
2
3 (define (snap t)
4   (cond
5     [(false? t) (list #false 1 #false)]
6     [else (define v (second t))
7           (if (odd? v)
8               #false
9               (list (snap (first t)) (+ v 1) (snap (third t))))]))
```

Let us compare the code in figure 1 with the OCaml version in figure 2. Instead of a comment, the OCaml snippet defines the `tree` type explicitly. The OCaml version of `snap` is nearly identical to the Racket version, except for the use of a pattern matcher[6] and numerous injections into the algebraic datatype. The former hides the projections out of the algebraic datatype and, via the introduction of pattern variables, greatly facilitates Hindley-Milner style type inference. The latter is something that "untyped developers" cite as an obstacle to using typed languages and that a migratory type system may not assume.

When Racket developers design functions such as `snap`, they reason about its possible inputs as sets. In this specific case, a developer knows that the first line in the conditional subtracts `#false` from the possible inputs. Hence `snap` has to deal with nothing but three-element lists in the second clause, meaning it is safe to extract the `second` item from `t`.

[6] Racket also comes with `match`. For the purpose of this illustration, we present the function in the style inherited from Scheme, especially because few scripting languages have pattern matching constructs.

Listing 2 The OCaml equivalent of `snap`

```
type tree = False | Triple of tree * int * tree

let rec snap t =
  match t with
  | False -> Triple(False,1,False)
  | Triple(left,v,right) ->
    if (v mod 2 <> 0)
    then False
    else Triple(snap(left),v+1,snap(right))
```

From the perspective of the type system, the developer asserts that the underlined occurrence of `t` in line 6 does not just belong to `Tree` but to `[list Tree Integer Tree]`. More generally, if Racket developers are willing to turn type-oriented comments into type declarations but want to leave function definitions alone during migration, the type checker must assign different sub-types to different occurrences of the same variable, depending on which type predicates govern the occurrence in the flow graph.

We re-use the term *occurrence typing* [29, 50] for this idea. While both Wright's Soft Scheme [56] and Flanagan's Spidey Scheme [20] incorporate simple, syntax-oriented variations on this idea, Typed Racket systematically incorporates it into the type judgment. Every function type comes with propositions that hold when the function returns `#false` or a non-false value, respectively. The type checker exploits these propositions and also combines them with propositions in the Racket code, mimicking the kind of propositional reasoning "untyped developers" perform on a daily basis [51].

Now consider the function `false?`, which `snap` uses to discriminate between different trees:

```
false? : [Any -> Boolean : #:+ False #:- (! False)]
```

It uses two optional annotations to articulate two propositions. Specifically, `#:+` says that if `(false? x)` returns `#true`, then `x` has singleton type `False`, i.e., `x` is `#false`; similarly, `#:-` tells the reader and the type checker that if the result is `#false`, then `x` cannot be `#false`.

Listing 3 The Typed Racket version of `snap`

```
(define-type Tree (U False [List Tree Integer Tree]))

(: snap (Tree -> Tree))
(define (snap t)
  (cond
    [(false? t) (list #false 1 #false)]
    [else (define v (second t))
          (if (odd? v)
              #false
              (list (snap (first t)) (+ v 1) (snap (third t))))]))
```

With this in mind, it is easy to see how a developer can migrate the code from figure 1 to the one in figure 3, *without modifying the function at all*. It suffices to define the `Tree` type and to declare the function's type. The occurrence type checker can reconstruct the types of the expressions in `snap` from just these two pieces of type information.

Typed Racket generalizes occurrence typing to higher-order functions, e.g.

```
filter : (All (a b) [ [a -> Any : #:+ b] (Listof a) -> (Listof b) ])
```

Naturally, `filter` is polymorphic. Furthermore, if its predicate argument specifies a positive proposition `b`, its result is a list of elements that satisfy `b`.

Occurrence typing also comes in handy for dealing with the numeric tower [41] that Typed Racket inherits from Racket and Scheme. This numeric tower allows developers to use numbers and operations on them the way mathematicians present them, with computer-based numbers mixed in for performance. Consequently, Racket code comes with numerous idioms that rely on mathematical sets instead of the disjoint numeric types based on machine arithmetic, commonly found in conventional languages.

Typed Racket's corresponding type hierarchy starts with `Complex` numbers, which contain the `Reals`. The latter subdivides into exact `Rationals`, including `Integers`, plus inexact, IEEE `Floats`. Within the `Integers`, Racket is aware of `Fixnum`, `Index`, and `Byte`. Finally, Typed Racket needs the type `Zero` because the value zero shows up in several different, disjoint sets and yet plays a special role in comparisons.

Listing 4 Typed Racket and the numeric tower

```
(: sum-vector [(Vectorof Integer) -> Integer])
(define (sum-vector v)
  (define n (vector-length v))
  (let loop ([i 0] [sum 0])
    (if (< i n) (loop (+ i 1) (+ sum (vector-ref v i))) sum)))
```

The types for numeric comparisons exploit occurrence typing to reify the numeric subsets:

```
> : [Real Zero -> Boolean : #:+ Positive-Real]
```

This type says that if `(> x 0)` produces `#true`, `x` has type `Positive-Real`. Additionally, reasoning about numeric idioms benefits from a lightweight form of intersection types:

```
+ : (case-> (Integer Integer -> Integer) (Float Float -> Float) ...)
```

This `case->` type lists function types and picks the first one that matches the use. Using such function types plus occurrence typing, Typed Racket can, for example, prove the type correctness of the `sum-vector` function in listing 4, including the fact that `i` is always a proper vector index. The language's optimizer may then safely exploit this fact [52].

Listing 5 Typed Racket and the numeric tower

```
(: transpose (All (a ...) [(Listof a) ... -> (Listof (List a ...))]))
(define (transpose . l)
  (if (andmap empty? l)
      '()
      (cons (map first l) (apply transpose (map rest l)))))
```

Like its popular cousins, Racket comes with a large variety of useful accommodations. For example, many Racket primitives take a variable number of arguments, e.g., `map`:

```
(map (lambda (s n) (- (string-length s) n)) '("hello" "world" "bye") '(3 4 1))
```

Of course, developers may also define such functions; see listing 5 for a simple illustration.

To support multi-variable functions in a sound manner, Typed Racket comes with variable-arity at the type level [43]. Here is the type signature for `map`:

```
map : (All (c a b ...)
        [ [a b ... b -> c] (Listof a) (Listof b) ... b -> (Listof c) ])
```

Like `filter`, `map` is polymorphic, but its type introduces an unbounded number of type variables to describe the number of lists that `map` traverses simultaneously. As listing 5 shows, Typed Racket is sufficiently powerful to type check uses of `map` via local type inference.

Finally, a lot of Racket code uses class-based, object-oriented programming, which comes with its own idioms [24]. Again like in other untyped languages, classes are first-class run-time values. Using those, Racket developers abstract over classes with functions and methods to construct just the right kind of class hierarchy at run-time. While this well-known "mixin" pattern has a long history, conventional type systems all too frequently identify classes with types and thus cannot deal with either mixins or other idioms using classes as values.

The current version of Typed Racket accommodates both function and method-based mixins, though it took several years to design a sound extension of the type system [47] and two more to implement and evaluate this design [46]. The extension rests on two theoretical novelties: (1) types for classes, for functions on classes, and so on; and (2) a contract system that ensures that class operations in untyped code respect the integrity of typed classes, mixins, etc. Practically the implementation must facilitate the addition of types to classes used in ordinary circumstances and make the protection of classes that flow across type boundaries reasonably efficient. For the former, see listing 6; for the latter, we point the reader to the original implementation paper [46, section 4] and our recent work on performance evaluation [45], both of which are discussed in the last section.

4 Type Soundness for Mixed-typed Programs

While industrial researchers may trade ideals for practical concerns, especially performance, academic researchers have the moral obligation to strive for them—because nobody else will and society affords them exactly this luxury with generous support. In the context of migratory typing, soundness is the critical ideal.

On one hand, we clearly want the usual soundness of fully typed programs; on the other hand, we also want a generalized notion for programs that link typed and untyped pieces. Formally, the usual soundness theorem states [57] that if a program type checks, running it can have exactly one of three possible outcomes (**MT1**):

1. the program execution terminates, returning and printing values of the predicted types;
2. the execution diverges;
3. the execution ends in one of a number of well-specified exception states. These exceptions are due to partial computational primitives such as division and indexing.

For un(i)typed[1] languages such as plain Racket, (MT1) holds if memory access is safe, but more computational primitives are partial than in a typed setting [14, part I(ch. 5)].

To generalize (MT1) to mixed-type programs, we must make an assumption about the linking of typed and untyped code and hence execution. Given our desire to run mixed-typed programs easily, we clearly do not wish to deal with the twin language as a truly foreign language. Instead we assume that all cross-boundary traffic uses the bit-level representation of values from the original, untyped language and specifies the type of the crossing value in the typed module. Now a value flowing across such a boundary may not meet the expectations expressed as its type, which requires adding one clause to the above three (**MT2**):

4. the execution ends in an exceptional state and points to one of the fixed number of boundaries between a typed piece of code and an untyped one. After all, each boundary represents a distinct, programmer-defined and partial computational primitive.

Let us inspect this generalized type soundness theorem from an operational perspective. As mentioned, a "migrating developer" who links a newly typed module to an untyped code base must add type specifications to all import statements so that the type checker can check their uses statically. Say an untyped function `f` is imported with `(D -> R)` imposed as its type. This type may not match the untyped reality, which is why the typed twin language must insert run-time checks to prevent certain problems. Here are some sample scenarios of how things can go wrong and how run-time checking works:

- `f` cannot cope with elements of `D`. In this case, the run-time checks of the underlying untyped language eventually catch the error and issue an appropriate message. Here it is critical that we assume the same bit-level representation for boundary-crossing values so that the primitives of the untyped function may use the tag bits for run-time checking.
- `f` does not produce elements of `R`. Since the soundness of the type checker depends on the `R`ness of the result value, (1) every `R` must come with a run-time check that can enforce `R`ness and (2) the type checker must insert this check at all call sites of `f`.
- `f`'s domain `D` is a function type, too, say `(D1 -> R1)`. When `f` is called, the typed code sends a typed function `g` into untyped code. In this case, applications of `g` must be protected so that all arguments are values in `D1`; type checking guarantees `R1`ness.

In all cases, the failure of a run-time check might be due to a bad type specification—that is, the untyped code cannot live up to the type imposed by the developer—or an error in the untyped code with respect to an expressed or implied type. And if things do go wrong, Typed Racket's exceptions come with a highly informative error message that points developers to the specific problem boundary and presents a witness value that explains the mismatch between the value and its type.

4.1 "The Dangers of Moral Turpitude" [35]

A type system that fails to satisfy (MT2) can resurrect all the problems of unsafe languages such as C++, which fails to live up to plain type soundness (MT1). These failures have serious implications for both programmers and ordinary users. Recall that C++ checks types but executes programs without enforcing the interaction between the untyped run-time system and the type-checked code. Hence, C++ programs interpret operations on bit patterns regardless of whether it is appropriate to apply the operation to the data that these bits represent. As long as the hardware does not object, the program execution continues. The program may seemingly terminate "normally," printing all kinds of output on the way. Alternatively, the misapplication of the operation may eventually trigger an interpretation of bits to which the hardware objects, resulting in a `segfault` long after the original misinterpretation.

To understand how a failure of (MT2) can trigger the first kind of problem, consider the module in listing 6. It exports a class that represents a simplistic voting machine. An importing module instantiates the class with a list of candidates:

```
(define my-voting-machine
  (new voting-machine% [candidates '("DonaldDuck" "HolyCow")]))
```

Once the votes in a district are tallied, the user can call the `add-votes-from-district` method to consolidate the tally with the running total:

```
(send my-voting-machine add-votes-from-district
  '(("DonaldDuck" 2) ("HolyCow" 4)))
```

This call adds two votes to the total of `"DonaldDuck"` and four for `"HolyCow"`.

Listing 6 A Voting Machine and the Lack of Sound Linking

```
#lang typed/racket

(provide voting-machine%)

(define-type Count {List String Natural})
(define-type Tally {Listof Count})

(define voting-machine%
  (class object%
    (super-new)

    (init {candidates : [Listof String] '()})

    (field [votes : Natural 0]
           [tally : Tally
            (map (lambda ({s : String}) (list s 0)) candidates)])

    (define/public (show)
      (sort tally second-of-pair->))

    ;; names not on delta did not get any votes,
    ;; names on delta but not on tally field are "write ins"
    (define/public (add-votes-from-district {delta : Tally})
      (for ((district-count : Count delta))
        (define-values (name delta) (apply values district-count))
        (define old-count (assoc name tally))
        (set! tally
              (match old-count
                [`(,_name ,old)
                 (define new (+ delta old))
                 (cons (list name new) (remove old-count tally))]
                [#false (cons district-count tally)])))))
```

Now consider the following method call:

```
(send my-voting-machine add-votes-from-district
  '(("DonaldDuck" 2) ("RonnyM" 3) ("HolyCow" -1)))
```

Given that the type signature of the method demands a *natural* number for vote counts, this call cannot type check in a typed module. Imagine what would happen, however, if an *untyped* module initiated this call. If (MT2) holds, the boundary between this client module and the module from listing 6 checks that all vote counts are exact, non-negative integers and therefore signals an error when it encounters `-1`. If (MT2) does *not hold*, no such check is run and `"HolyCow"` loses votes and ends up trailing `"DonaldDuck"`. The program execution goes on, and nobody may ever know about the invalid vote subtraction.

A failure to implement (MT2) can also trigger C++-style segfaults if the compiler for typed modules exploits type information for code generation. For example, it may insert an integer multiplication yet the lack of run-time checks may allow floats to flow into this operation. In short, unsound linking really introduces the whole range of problems that the creation of high-level languages—typed and untyped—aims to eliminate for good.

5 Keeping the Cost of Run-time Checks Low

So every single time a (semantic) value flows from an untyped piece of a program to a typed one, a run-time check ensures that the value lives up to the specified type. A flat value, say a number, requires the same kind of checking that any sound, untyped language performs. When a higher-order value such as a function or an object crosses a boundary, checking a type-like property is impossible because, semantically speaking, these values are infinite. Hence the run-time system delays the relevant checks until the function is applied, a message is sent to an object, etc [19]. Finally, for a compound value, e.g., an array, the run-time check either inspects every element, even if none are actually accessed, or delays the checks until an access is executed. For mutable values, this latter strategy is imperative.

None of these checks come for free. While the checks for flat values are relatively inexpensive for a single crossing, the cost for other checks is non-trivial for single crossings and especially for high-frequency crossings. In Typed Racket, these checks impose two kinds of costs: the allocation of wrappers for delaying the checks and the time for the delayed run-time checks. To make this latter cost concrete, imagine a typed function that flows into an untyped part of the program. If the untyped part applies the function a million times, the argument checks kick in that often. If the already-wrapped function flows back out of typed territory into different untyped code, the boundary check wraps it again, and every application must penetrate two layers of wrapping and execute two delayed checks.

Due to this anticipated cost of boundary crossings, the Typed Racket vision paper [49] argues for reasonably large units of migration. Specifically, it argues that Racket modules hit the sweet spot between the desire to migrate code easily and to keep the run-time cost low.

On one hand, the point of modules is to bundle many functions into one unit, making some visible to client modules, hiding others. An exported function performs a decent amount of work before it hands control back to the client. It may call several hidden functions, usually in a hierarchical manner. By contrast, an individual function often connects to the surrounding context via numerous free identifiers, and the flow of control may cross this boundary much more often than a module boundary. In short, the fewer boundaries the fewer crossings are to be expected, which translates into a lower cost of run-time checking.

On the other hand, most Racket modules are reasonably small. With few exceptions, the modules in known applications consist of several hundred to a couple of thousand lines of code. Although adding types to such modules is clearly much more work than adding types to a single function, the paper assumes that developers work at the level of complete modules and that adding types to a module is a reasonable amount of work. Furthermore, if the IDE comes with approximate type inference tools, developers can rely on those to reduce their work to checking, and correcting, inferred types.

6 Related Work

Migratory typing is one particular instance of the 35-year old idea of *optional typing* for untyped languages, first found in Common Lisp [42]. The basic idea of optional typing is to allow the explicit specification of type information but not necessarily enforce it. Compilers and other development tools may then exploit this information in an appropriate manner. Both StrongTalk [9] and pluggable type systems [8] fall into the same category.

Modern incarnations include industrial and quasi-industrial systems such as Hack,[7] Flow,[8] TypeScript,[9] StrongScript[36], and Typed Clojure [7].

Optional type systems usually, but not always, satisfy (MT1) and most fail (MT2). As a result, many suffer from some of the flaws spelled out in section 4.1. Even though we understand the constraints of their creators as mentioned above and re-iterated below, the situation nevertheless suggests to us that Bertrand Russell's motto of "the advantages of theft over honest toil" has been applied to soundness.

Contemporaries of migratory typing, *hybrid typing* [21] and *gradual typing* [37, 38, 39] are theoretical designs that satisfy both (MT1) and (MT2). The teleology and the technical details differ from design to design. Hybrid typing aims to increase the power of static type checking. It allows programmers to add arbitrary predicates to type specifications, which are checked statically as much as possible and dynamically otherwise. In comparison, gradual typing aims to put static and dynamic typing on equal footing without violating soundness. It thus allows programmers to add type information on a purely optional basis *anywhere* in a program and inserts casts automatically as needed. Finally, the purpose of migratory typing is to support the migration of code from an untyped setting to a typed one, while preserving the ability to run any mixed-typed software system with the same guarantees as the fully untyped or fully typed ones. Clearly gradual typing can be used for migrating code in a sound manner, but it is equally well suited for annotating extremely small fragments with types for documentation of logical invariants or for exploratory coding in the context of a fully-typed system. By contrast, industrial optional type systems also aim to assist with the migration of code but accept temporary or even permanent unsoundness in the process.

Besides Typed Racket, which has been in development for the past ten years, two other academic groups have started efforts to implement optionally typed systems that satisfy (MT1) and (MT2): Reticulated Python [53] and a gradually typed Smalltalk [3]. Both systems are used to experiment with casting strategies [40], including strategies that give programmers some control over the interchange of values [4]. It would be interesting to assess the performance of these systems with the same metrics as Typed Racket; see below.

7 Assessing Typed Racket, Lessons for Language Designers

Numerous Racket developers have embraced Typed Racket over the last ten years, including people in the Racket development team. We, the Typed Racket developers, have used Typed Racket extensively, in many different scenarios and situations. While some of the uses correspond to those imagined ten years ago, developers equally often just *add* typed modules to existing applications or create entire new libraries in Typed Racket, e.g., `math`, `pict3d`, `whalesong`. When developers use Typed Racket in this manner, they may use Racket idioms but they may also employ idioms they know from statically typed languages, in which case our type system is typically not what they expect.

We base the following assessment of Typed Racket on this usage history. The assessment accounts for all three principles from section 1 plus the overall goal of assisting developers.

Programming Idioms Typed Racket's type system mostly succeeds in accommodating the idioms of untyped Racket. Migrating *mostly-functional* Racket modules requires a relatively small amount of work. For most such programs, it usually suffices to add type definitions that

[7] See `hacklang.org`, last visited 18 Mar 2017.
[8] See `flowtype.org`, last visited 18 Mar 2017.
[9] See `typescriptlang.org`, last visited 18 Mar 2017.

name (recursive) union types and type declarations for functions, variables, and structure fields. Local type inference reduces the burden of adding type declarations in some cases.

Recurring complaints in this setting concern uses of (first-class) polymorphic functions and a lack of (some) refinement typing. When it comes to polymorphic functions, the existing local type inference algorithm fails too often, forcing developers to insert explicit type applications into existing code. Not surprisingly, Racket developers also have a certain amount of "refinement typing" in mind. This observation is the motivation of our recent investigation of refinement typing for Typed Racket [27], an extension that we will soon merge into the production branch.

By contrast, the migration of *object-oriented* Racket modules demands significantly more effort than the migration of functional code, requiring both significantly larger type annotations and many more touch-ups of existing code. For example, developers must write down the types of classes separately from the code for classes—except for those used in a strictly first-order fashion—causing a high degree of redundancy. Similarly, the type system is still missing occurrence typing for fields, which may trigger rewrites of method code.

In general, code migration demands interventions for about 5% to 20% of the existing lines of code. For functional code the number ranges in the lower part of the interval (5% to 10%), down from about 10% to 12% for the first implementation of Typed Racket. Object-oriented code needs interventions for 15% to 20% of the original lines. The difference is partly due to the programming idioms and partly due to maturity of the functional type system, which has been in development since 2007, while the object-oriented one has a mere five-year history.

This experience suggests an important, often overlooked lesson for language designers. *Syntactic engineering* must be taken into account from the very beginning to make the new language constructs as convenient to use as possible. Once developers experience a lot of friction, they might be reluctant to stay the course or take a second look.

The Benefits of Soundness The system lives up to all expectations that developers have of sound language implementations. The static type checking eliminates many dynamic errors that Racket code suffers from. The newly added dynamic errors resemble the usual run-time exceptions of statically typed programs; they also always point to the boundary where the expectations of a typed module clash with the realities of its untyped surroundings.

Although the implementation comes with the usual errors, the semantic model tends to clarify how to fix these problems quickly. One exception concerns exported parametrically polymorphic functions. To enforce parametricity, the contract system inserts wrappers that subtly change the behavior of the functions and thus the entire program. We do not know how to solve this problem efficiently.

The Cost of Soundness Performance has emerged as a major problem area over the past couple of years, justifying our original concern about the run-time cost of the inserted dynamic checks. Early indications included posts to our mailing list describing scenarios where mixed-typed programs exhibited abysmal performance. In some cases, developers found satisfactory work-arounds; in others, they solved the problem by abandoning Typed Racket or adding types to all of the code. After analyzing these problems on an *ad hoc* basis for quite some time, we decided to develop an evaluation method.

Our evaluation method [25, 45] calls for measuring *all possible combinations of typed and untyped modules* for every benchmark program. If a program consists of n modules, there are 2^n such configurations, implying an exponential effort. After extensive evaluations, we can now confirm that sampling a *linear* number of configurations suffices [25]. Hopefully this confirmation will encourage others in the field to use the evaluation method.

Applying the method to Typed Racket shows that mixed-type configurations suffer from a huge overhead. Some types compile to expensive run-time checks, and others allocate a lot of memory. Worst, some configurations call functions many times across boundaries.

Applying the method to Reticulated Python[10] suggests that it may suffer from similarly disabling performance problems. Due to Reticulated's transpiler, the performance lattice consists of $1 + 2^n$ programs: the original Python program and 2^n Reticulated configurations. Our measurements suggest a 2x average slow-down from plain Python to unannotated Reticulated program alone and they show that, as types are added, Reticulated inserts additional casts and the programs experience additional slow-down. Nevertheless, Vitousek et. al [54] conjecture that the overhead of Reticulated is an order of magnitude smaller than Typed Racket's—though they do so without having applied the evaluation method.

These performance evaluations suggest three lessons. First, soundness is an ideal that does not come cheap. If academic researchers wish to convince their industrial colleagues that soundness of optional type systems is a feasible idea, they must develop (1) suitable evaluation methods from the get-go and (2) pay attention to performance at every stage. Second, retro-fitting implementations is hard. The Typed Racket team is now exploring alternative implementation strategies, especially a just-in-time compiler that can take advantage of the run-time checks [5]. How much this alternative implementation can reduce the overhead of migratory or gradual typing remains an open question. Third, some readers may jump to the conclusion that our industrial colleagues who trade soundness for performance are correct after all. We consider this conclusion premature. As mentioned, we accept the moral obligation of pursuing an ideal until there is conclusive proof of failure but an evaluation of two implementations does *not* disprove any hypothesis. We will continue to investigate sound migratory typing and its implementation until we know for sure that the failure is total.

Migrating Code As for the original goal—assisting software developers with maintenance via the migration from untyped to typed code—we also have to report mixed insights, all based on anecdotal evidence. On one hand, our experience tells us that migrating entire modules is rarely a problem because the size of Racket modules is (now) reasonably small. On the other hand, outside developers report that modules are too large for a migration effort. People would much prefer to add the type invariants of just a key algorithm to a module when they have to revisit the code. This desire suggests a preference for an approach based on gradual typing rather than migratory typing.

The contradictory evidence presents a research challenge. While we are aware of the literature on measuring the benefit of adding types to programs [28, 31], we have not yet figured out how to construct a similar, repeatable experiment on migratory typing from these results. Our ideal scenario would involve a comparative study of a reasonably large code base and several well-qualified developers; moreover, the development task should come with a proper incentive. In other words, we simply do not understand how student experiments on small programs are predictive of real-world behavior, which is what we ultimately aim for.

Until we have answers to both the performance challenge and the benefits question, the entire Typed Racket project remains speculative. While our own experience and anecdotal evidence seem to tell us that Typed Racket adds value to the eco-system of Racket, scientific evidence for these points remains elusive and calls for additional research. We conjecture that gradual typing and other attempts at sound optional typing face similar challenges, and we therefore consider the proposed research the highest priority for this entire area.

[10] Ben Greenman and Zeina Migeed are currently conducting this investigation.

Acknowledgements We thank Eli Barzilay, Ambrose Bonnaire-Sergeant, Robert "Corky" Cartwright, Ryan Culpepper, Earl Dean, Christos Dimoulas, Eric Dobson, Mike Fagan, Cormac Flanagan, Daniel Feltey, Philippe Meunier, Max New, Neil Toronto, Jan Vitek, and Andrew Wright for contributing at various stages to the ideas of typing untyped languages.

References

1 Alexander Aiken and Brian R. Murphy. Static type inference in a dynamically typed language. In *Principles of Programming Languages*, pages 279–290, 1991.

2 Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Principles of Programming Languages*, pages 163–173, 1994.

3 Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, 96(P1):52–69, December 2014.

4 Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. Confined gradual typing. In *Object-Oriented Programming Systems, Languages & Applications*, pages 251–270, 2014.

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

6 Hans J. Boehm. Partial polymorphic type inference is undecidable. In *Foundations of Computer Science*, pages 339–345, 1985.

7 Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical optional types for Clojure. In *European Symposium on Programming*, pages 68–94, 2016.

8 Gilad Bracha. Pluggable type systems. In *Object-Oriented Programming Systems, Languages & Applications*. Companion (Workshop on Revival of Dynamic Languages), 2004.

9 Gilad Bracha and David Griswold. Strongtalk: typechecking Smalltalk in a production environment. In *Object-Oriented Programming Systems, Languages & Applications*, pages 215–230, 1993.

10 L. Cardelli. Typeful programming. Technical Report 45, Digital Systems Research Center, May 1989.

11 Robert Cartwright and Mike Fagan. Soft typing. In *Programming Language Design and Implementation*, pages 278–292, 1991.

12 William Clinger and Jonathan Rees. The revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.

13 Mike Fagan. *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages*. PhD thesis, Rice University, 1991.

14 Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.

15 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The DrScheme project: An overview. *ACM SIGPLAN Notices*, June 1998. Invited paper.

16 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The TeachScheme! project: Computing and programming for every student. *Computer Science Education*, 14:55–77, 2004.

17 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket Manifesto. In *First Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 113–128. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

18 Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *J. Functional Programming*, 12(2):159–182, 2002.

19 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, pages 48–59, 2002.

20 Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, 1997.

21 Cormac Flanagan. Hybrid type checking. In *Principles of Programming Languages*, pages 245–256, January 2006.

22 Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Programming Language Design and Implementation*, pages 235–248, 1997.

23 Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *Programming Language Design and Implementation*, pages 23–32, May 1996.

24 Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Asian Symposium on Programming Languages and Systems*, pages 270–289, 2006.

25 Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. Performance evaluation for gradual typing. *J. Functional Programming*, 2016. Submitted for publication.

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

27 Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. Occurrence typing modulo theories. In *Programming Language Design and Implementation*, pages 296–309, 2016.

28 Sebastian Kleinschmager, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. Do static type systems improve the maintainability of software systems? An empirical study. In *International Conference on Program Comprehension*, pages 153–162, 2012.

29 Raghavan Komondoor, G. Ramalingam, Satish Chandra, and John Field. Dependent types for program understanding. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *lncs*, pages 157—173, 2005.

30 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM Transactions on Programming Languages and Systems*, 31(3):1–44, 2009.

31 Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Object-Oriented Programming Systems, Languages & Applications*, pages 683–702, 2012.

32 Philippe Meunier. *Modular Set-Based Analysis from Contracts*. PhD thesis, Northeastern University, 2006.

33 Philippe Meunier, Robert Bruce Findler, and Matthias Felleisen. Modular set-based analysis from contracts. In *Principles of Programming Languages*, pages 218–231, 2006.

34 D. Rémy. Typechecking records and variants in a natural extension of ML. In *Principles of Programming Languages*, 1989.

35 John C. Reynolds. Types, abstraction, and parametric polymorphism. In *IFIP Congress on Information Processing*, pages 513–523, 1983.

36 Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete types for typescript. In *European Conference on Object-Oriented Programming*, pages 76–100, July 2015.

37 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop, University of Chicago, Technical Report TR-2006-06*, pages 81–92, September 2006.

38 Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, pages 2–27, 2007.

39 Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Dynamic Languages Symposium*, pages 1–12, 2008.

40 Jeremy G. Siek, Michael M. Vitousek, and Shashank Bharadwaj. Gradual typing for mutable objects. Unpublished manuscript, available at `http://ecee.colorado.edu/~siek/gtmo.pdf`.

41 Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. Typing the numeric tower. In *Practical Aspects of Declarative Languages*, pages 289–303, 2012.

42 Guy Lewis Steele Jr. *Common Lisp—The Language*. Digital Press, 1984.

43 T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical variable-arity polymorphism. In *European Symposium on Programming*, pages 32–46, March 2009.

44 Norihisa Suzuki. Inferring types in Smalltalk. In *Principles of Programming Languages*, pages 187–199, 1981.

45 Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *Principles of Programming Languages*, pages 456–468, 2016.

46 Asumu Takikawa, Daniel Feltey, Sam Tobin-Hochstadt, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. Towards practical gradual typing. In *European Conference on Object-Oriented Programming*, pages 4–27, 2015.

47 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Object-Oriented Programming Systems, Languages & Applications*, pages 793–810, 2012.

48 Satish Thatte. Quasi-static typing. In *Principles of Programming Languages*, pages 367–381, 1990.

49 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Dynamic Language Symposium*, pages 964–974, 2006.

50 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Principles of Programming Languages*, pages 395–406, 2008.

51 Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *International Conference on Functional Programming*, pages 117–128, 2010.

52 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Programming Language Design and Implementation*, pages 132–141, 2011.

53 Michael M. Vitousek, Jeremy G. Siek, Andrew Kent, and Jim Baker. Design and evaluation of gradual typing for Python. In *Dynamic Language Symposium*, pages 45–56, 2014.

54 Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big types in little runtime. In *Principles of Programming Languages*, pages 762–774, 2017.

55 Mitch Wand. Finding the source of type errors. In *Principles of Programming Languages*, pages 38–43, 1986.

56 Andrew Wright. *Practical Soft Typing*. PhD thesis, Rice University, 1994.

57 Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, pages 38–94, 1994.

58 Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. In *Lisp and Functional Programming*, pages 250–262, June 1994.