Why are Rust and TypeScript more liked than C++ and JavaScript?

1 The Pragmatics Question

Time and again, a start-up picks an untyped programming language, say JavaScript or Python, to get off the ground quickly, including languages lacking in type-notation expressiveness, say C++. Rumor has it that such languages are good for the initial building process. Yet, as the company grows and recruits additional software developers, the lack of types or type expressiveness becomes noticeable.

People then recognize that typed languages facilitate the "maintenance" of code. As the preceding chapter explains, explains type systems—sound and unsound—can assist developers in a range of work situations, including what is commonly called maintenance. Properly understood, maintenance denotes variants of every single work situation: adding code means design and code creation; it means adapting the code base to cope with new forms of data; it means equipping the code base with additional functionality; it means documenting the interfaces of components; and so on. At each stage, types and type checking helps.

Without checked types, the on-boarding process becomes a painful exercise. New developers do not know which kinds of objects come with which methods; they have to look at the component's informally documented interface or the code, because the IDE cannot display a matching menu. They do not know the signatures of these methods, and once again, the IDE cannot help with a method-call template. The informal nature of component interfaces poses another problem: even though these informal signatures make up an appropriate use of comments, they easily get out of sync with the code. When the underlying machine finally signals a mistake because some primitive computational operation fails, these new developers may have to search for the bug in their own code and in the *old* code.

Concretely, these problems show up in different shapes and forms. Most programmers know that JavaScript illustrates these problems well. Even with all the enhancements of the last two decades in place—especially the addition of classes and modules, plus validation modes—developers struggle with adding functionality to its code base. Developers embrace its typed sibling—TypeScript—as soon as it appeared.

Another example is C++. It comes with a type notation and a type checker, though the former lacks expressive power and the latter is unsound. While the preceding chapter indicates in which work situations sound type checking helps, the type notation's lack of expressive is an altogether different problem. Since C++ is a language that permits developers to directly manipulate memory addresses—known as pointers—you might expect that the type system enables developers to articulate their thinking about pointers. For example, the type notation should allow the creator of one component to communicate to the creator of another who is responsible for marking an address as no longer in use, if both have access to it. Similarly, it should empower developers to articulate which of two concurrent components may modify the content of an address. The C++ language lacks this power, however. Unsurprisingly, Rust is taking this area of the software development by storm, precisely because its type system has the appropriate expressive power.

In short, the emergence of TypeScript and Rust demonstrate how language creators can address these pragmatic problems of working developers. The key is to come up with sibling language that (1) comes with an expressive type systems and (2) supplies an interface for the smooth integration of code in *both* languages. This chapter covers the relatively simple case of combinations that mix untyped and typed languages, such as JavaScript with TypeScript. For the case of strengthening type systems, say C++ and Rust, please consult the relevant literature or blogs.

1.1 Design Choices

Language researchers recognized many decades ago how untyped code disadvantages developers as a code base grows. Since then, they have come up with a number of solutions. Here are three distinct approaches with different motivations:

- Going back to the 1980s, the Common Lisp language implements an *optional typing* approach. A developer may declare types in various places, but they are essentially hints to the compiler or the IDE. In other words, these tools trust any annotations and use them as appropriate.

- Historically speaking, the idea of *soft typing* comes next. A soft-type system supplements an existing language syntax. It typically does not come with a type notation that developers can use. Instead, it infers types and presents the results in a notation that the soft-type system creator considers appropriate. A soft-type system may still come with tools that assist developers with, say, understanding why a function supposedly has a particular domain or range type.

- *Migratory Typing* is the most recent and the most successful approach at the moment. Given some untyped language, language creators create a sibling language that is syntactically similar to the original one but comes with a type system. Indeed, once the type annotations are removed from a program in the sibling language, the result *is* a valid code in the original language.

You may wish to consult the literature and the blogosphere for additional ideas in this realm.

What makes migratory typing so successful is a synergistic combination of factors. The adjective "migratory" tells us that it is all about moving from one place to another. First, language creators wish to enable a smooth migration of code from the untyped world to the typed one. When software developers notice that a growing code base suffers from a lack of types, they may wish to migrate existing any piece of code that needs to be modified. This piece might be an expression or a module, but no matter what, the type system should pose as few obstacles as possible. Put differently, the type system should accommodate the idioms of the original, untyped language—for the ease of code migration from one language to another.

Second, as already mentioned, when a language has been in use for decades, software developers have created a large body of re-usable components, frameworks, and libraries. They will want a simple way of accessing these untyped components from newly created typed code, and they want to make sure that this access is in sync with the type annotations. A common solution is to provide a mechanism for specifying type interfaces for untyped code in a *post hoc* manner.

Finally, language creators also apply "migrate" to the software developers themselves. Imagine developers who have spent many years or even decades working with the untyped language. Although they may not want to migrate existing and working code it is easy to imagine that they want

Experienced developers know that touching code often means enbugging it.

assistance from a type system. Since these developer are used to certain idioms, one non-starter would be to force them to change their coding style just to accommodate the type checker. Once again, the type system must accommodate the idioms of the original, untyped language as much as possible—for the sake of developers.

The creators of Typed Racket and, a decade later, TypeScript recognized the critical nature of these factors and took them as objectives of their respective designs. Unsurprisingly, the resulting type systems resemble each other in some ways, and they differ in others. Both achieved the first and third objective in a reasonable manner, relative to the starting points. When it comes to the second one, however, the two designs differ radically.

When it comes to interactions between typed and untyped code, Typed Racket aims to satisfy the academic goal of achieving the highest degree of soundness. By contrast, the TypeScript documentation advertises that the language is essentially just JavaScript. More precisely, the two language designs implement two radically different semantic choices:

- The TypeScript approach is to validate that the types accurately describe the code in the typed component and, once this is done, to throw away the types. Clearly, this approach is *unsound*, because typed code interacts with untyped, and thus unchecked, code.

- The Typed Racket approach is to validate the typed component using the *post hoc* type specifications for an untyped component as given. Next it compiles these *post hoc* types into run-time checks at the boundary between typed and untyped pieces. If these checks discover a non-conforming value, this value is a witness to a concrete violation of the specified type. To alert the developers, the semantics assigns an error to the combined system .

Both of these solutions have distinct costs and benefits, for both software developers and language creators. It's time to consider those next.

1.2 Costs and Benefits

A migratory typing system shares the goals of a plain static type system, but it is clearly more complex than the latter. On the design side the complexity mostly shows up in the derivation rules, but, if a high degree of soundness is desirable, the translation from types to run-time checks requires quite some sophistication to keep the cost in check. On the usage side, the complexity appears in two forms. Even though the design of the type system aims to accommodate the idioms of the underlying untyped language as much as possible, developers still face a steep learning curve to get their code to pass the type checker. Furthermore, when developers impose a typed interface on existing untyped code, they invariably make mistakes, and someone has to cope and fix these mistakes later. Let's take a close look at the two sides.

From the perspective of a software developer, a migratory type system means that each piece code could live in either world: typed or untyped. As long as the developer works in the typed world, a migratory type system comes with all the same costs and benefits as a plain static type system: a decrease in expressiveness, support from the type system during most work situations, and so on. It all differs as soon as both languages are involved, and the difference has two major aspects.

The first difference concerns the interaction between typed and untyped code. If a developer wishes to use an untyped framework in typed code, it does not suffice to import it without ado. Otherwise the type checker does not know how to validate calls into the framework and returns from there, not to speak of higher-order uses. That is, the developer must find or create a typed interface for the imports from the framework.

Since these useful frameworks have typically evolved over a long time, they tend to consist of large amounts of code, and their creators may be retired or even deceased. The combination of these two factors means that the creation of such typed interfaces is a significant amount of labor. Both factors also make it difficult to come up with *correct* type signatures for the exported pieces of functionality. As a result, such *post hoc* interfaces tend to come with numerous errors, that is, with types that fail to accurately and correctly describe the presumably well-tested and exercised framework code. For an illustration of this point, find the on-line repository called, *Definitely Typed*, which is a collection of typed interfaces for JavaScript libraries that are used in TypeScript code. Visit some of the files to see how large they are. And check out the GitHub issues—closed and open—to get a sense of how many errors people have already discovered.

Bugs in these types may manifest themselves long after the release of their containing interface (files and components). Let's make this precise. Suppose the framework supplies a function that supposedly returns an integer but, for the given inputs, actually produces a string. Nothing prevents such an event, because the type checker does *not* validate untyped code. What happens next depends on whether the combined language comes with (1) run-time checks at the boundary, (2) run-time checks in the underlying semantics, and (3) none of these. In all cases, the task of discovering the bug and locating it, is left to the developer who imports the framework—and this is usually neither the developer of the framework nor the developer of its typed interface. Obviously, the very possibility of such events happening must count as a potentially large cost.

The second difference between a migratory type system and a plain static one is about expressive power. While it is true that types restrict what a programmer can say, a migratory typed system offers the best of both worlds in this regard. If a developer notices that the type checker cannot validate a particular kind of code, an escape to the untyped world solves the problem. Escape means, of course, that the developer must create an interface between the two pieces. But, unlike in the case of existing libraries or frameworks, in this case the creator of the typed code, the untyped code, and the interface between them is one and the same person. In short, the mixing of typed and untyped worlds via closely related programming languages offers a large benefit when it comes to the ease of turning ideas into working code.

From the perspective of the language creators, the cost/benefit analysis looks somewhat different. As mentioned, the creators must come up with judgment derivation rules that accommodate existing idioms and, if they want a sound type system, they must also develop a translation from types to run-time checks.

Before the language creators can even develop a type notation—not to mention derivation rules or type checkers—they must spend a good amount of time studying how programmers have used the untyped language in the past. These programmers don't just create code in a haphazard fashion, hoping for the best. With any training, experience, or both, they design code in a type-directed manner, just like the developers who use a typed language. But, they are not constrained by a specific type notation or by specific rules. Instead, they tend to think of sets of values and this thinking can be recognized as type-like reasoning. Hence the designers of migratory type systems must ask what "expressiveness" and "effectiveness" mean in this context, and how to balance the two. Unsurprisingly, this problem is by far the most complex language design problem mentioned in this book.

The semantic design choice—trust types or enforce them with run-time checks—also poses a significant problem for language creators. No matter what they choose, the result deeply affects the work of software developer:

- If they choose to throw the types away after validating them, they shift work to the software developers. As explained, the mix of typed and untyped code is going to cause mismatches between the types that specify an boundary and the values that cross the boundary. The continuing computations are erroneous in an insidious manner. Sometimes the problem is easy to discover, sometimes it takes quite a while for people to recognize it. Since the unprotected boundary provides no information about the mismatch, the work of discovering where the mismatch is located, and how to fix it, is completely left to the developer.

- If the language creators choose to compile types to run-time checks, they clearly help developers with the discovery of bugs, but at the same time, they create a different problem for the working developers. Every run-time check costs time and thus reduces the performance of the program. Every run-time check may disable compiler optimizations, costing additional time, space, and energy. A software developer working on performance-critical code may not be able to tolerate this kind of performance loss.

In sum, the design and use of a migratory type system comes with potentially large costs and benefits. This chapter presents a model of both a sound and unsound system; doing so naturally builds on the preceding chapters. Once these designs are understood, we return to the discussion of the above points in section 7. By contrast, the design of an accommodating type system is beyond the scope of this book; the next section supplies a taste of what it means, without explaining how the type system works.

2 A Type System for Untyped Languages

The preceding section explains that a migratory type system must satisfy two criteria: (1) It has to accommodate the idioms of the underlying, untyped language, and (2) it has to come with a mechanism for smoothly integrating untyped code into typed code and vice versa. This section covers the two topics with two corresponding subsections, using Racket and Typed Racket for this presentation. While the first one is somewhat orthogonal to the story of this book, the second one motivates the essential elements of the models presented in the remaining sections of this chapter.

2.1 A Type System for Untyped Idioms

Existing Racket code repositories validate that Racket programmers mostly think of types as sets of values. Their reasoning reflects naive set theory. If they know a bit of advanced type systems, their signatures (in comments) may use an existential quantifier to hide a type inside of a region; they may resort to a universal quantifier to appeal to parametric polymorphism; they may add dependencies between domains and ranges of functions; and so on. Typed Racket accommodates most of these ideas, at a mostly reasonable cost to the developer, as this section illustrates.

```
#lang racket                    #lang typed/racket

;; A TLC is one of:             (define-type TLC
;; - "red"                        (U "red"
;; - "green"                         "green"
;; - "yellow"                        "yellow"))

#; (TLC -> TLC)                 (: switch (TLC -> TLC))
(define (switch cc)             (define (switch cc)
  (cond                           (cond
    [(equal? "red" cc)              [(equal? "red" cc)
     "green"]                        "green"]
    [(equal? "green" cc)            [(equal? "green" cc)
     "yellow"]                       "yellow"]
    [(equal? "yellow" cc)           [(equal? "yellow" cc)
     "red"]))                        "red"]))
```

Figure 67: Finite sets of values are types

The most basic way of “thinking of types as sets” reduces to an enumeration of a finite number of elements. Say a software developer’s charge is to develop a controller for a traffic light that at any moment illuminates either its red bulb, its green one, or its yellow one. Put differently, the state of the traffic light is equivalent to its illuminated color. This analysis suggests that a natural data representation of the traffic light is the finite set containing the three obvious strings: "red", "green", and "yellow". As the left column of the figure 67 shows, the conscientious Racket developer writes down this choice using structured English. The module in the figure also comes with the switch function, which picks the bulb to be illuminated next.

A properly trained developer would replace these strings with constant names in both the types and the code.

The right column displays a conversion of this untyped module into a typed one. First notice how the type definition directly mirrors the informal data representation with an actual union type (U) of three constants. Second, the second use of a comment—the type signature of the switch function—also has an immediate and easy-to-understand representation in Typed Racket. Third, the body of the function definition remains the same, because the type checker can validate the function’s signature against the original code. That is, the type system allows a direct expression of data definitions as type definitions, and the type checker easily copes with function definitions for such simple examples.

Basic set operations make an appearance in almost every piece of code, and the type system accommodates the most important ones, as the next example demonstrates. Figure 68 presents two versions of the factorial as a conscientious Racket programmer, or a properly instructed novice, would write them. On the left we see the factorial function in plain Racket. It comes with two appropriate comments: a signature and a mathematical purpose statement. While the function definition itself matches the proper mathematical definition, note that the module comes with a basic unit test (written as a submodule). The right column of the figure presents a conversion of the left-hand code into a Typed Racket module.

In Racket "!" really is a function name.

```
#lang racket

;; Natural -> Natural
;; n! = 1 · 2 · ... · n
(define (! n)
  (if (= n 0)
      1
      (* n (! (- n 1)))))

(module+ test
  (require rackunit)
  (check-equal? (! 3) 6))
```

```
#lang typed/racket

(: ! (Natural -> Natural))
;; n! = 1 · 2 · ... · n
(define (! n)
  (if (= n 0)
      1
      (* n (! (- n 1)))))

(module+ test
  (require typed/rackunit)
  (check-equal? (! 3) 6))
```

Figure 68: Set differences between types confirm case-based reasoning

This second complete example already demonstrates some important points. First, and as before, the function definition does not have to change. As long as the programmer specifies a type that matches the function's code, the code can often remain the same. Second, the changes to the notation are simple. The programmer adds typed/ to the language-specification line and the require specification for the unit-test library. Next, the signature (comment) is turned into a type specification that associates the name of the function ! with its type (Natural –> Natural). Third, the type copies Natural from the unchecked signature, and the type checker successfully validates this type against the function definition.

A use Natural—instead of the conventional Integer—illustrates how programmers can reason in an untyped language. In the context of an ordinary statically typed language, a programmer would use Integer to specify the inputs and outputs of the factorial function, even though every discrete mathematics textbook says that it is a function on **N**. A programmer who uses an untyped language can go along with the textbook, as the signature on the left of figure 68 shows. The signature informs readers that ! must be applied to natural numbers, and the creator of ! must validate this constraint for the recursive call. Since n is larger than 0 in the else branch of the conditional, (- n 1) is still a natural number, meaning applying ! to this number is acceptable.

Racket's type checker reasons in a similar manner. To start with, the type system defines Natural as 0 or $n + 1$ for a natural number n. Furthermore, Racket's migratory type system employs set-based reasoning via basic logical propositions. Thus, while n has type Natural in the conditional check, (1) it can focus on 0 when the outcome of the check is positive, and (2) it can subtract 0 from the type for the else case. Hence, by the time it checks the framed expression, it may conclude that (1) n is positive and (2) (- n 1) is still in Natural. In general, the type of a variable depends on where it occurs within a nest of conditionals and whether the type checker can convert conditions into propositions that narrow down the specified type.

Figure 69 illustrates this principle with a higher-order example. The figure again presents two columns with untyped and typed code. The framed expression within the typed module is the focus of this example.

Take a look at the untyped module. Its purpose is to represent and process shapes, concretely a mix of circles and squares. A structure definition is used to represent basic shapes; an informal data definition ties them together into the (set of all) Shapes. The main function, area-of-just-squares, consumes a list of Shapes and produces the area of all squares on this list. It employs several higher-order functions: filter, map, and foldr. Of these, the first one whittles the given list of arbitrary shapes down to a list of just squares. Hence, the programmer can use map to apply a function for computing the area of one square to all elements of this list—without any fear of accidentally applying the structure-selector to the wrong kind of struct.

Although this code does not contain any conditional per se, it nevertheless demands a form of conditional reasoning from the programmer. Technically, the reasoning lifts the hidden conditional from filter and combines it with square?, the given predicate. By combining the two, a programmer may conclude that the remaining occurrence of sq* in the definition of area-of-just-squares has a narrower type than [Listof Shape], namely [Listof Square].

Now take a look at the typed module on the right side. This conversion requires a bit more effort than for our first example, mostly due to

```
#lang racket

#; (Shape = Circle or Square)

(struct circle (radius))
#; (circle r) ; r is the radius
;; r is REAL

(struct square (side))
#; (square s) ; s, a REAL, is
;; the length of its side

#; ([Liston Shape] -> REAL)

(define (area-of-just-squares s)
  (define sq* (filter square? s))
  (define ar* (map sq-area sq*))
  (define sum (foldr + 0 ar*))
  ;; return:
  sum)

#; (Square -> REAL)
(define (sq-area x)
 (sqr (square-side x)))
```

```
#lang typed/racket

(define-type Shape (U Circle Square))

(struct Circle ((radius : REAL)))
#; (circle r) ; r is the radius


(struct Square ((side : REAL)))
#; (square s) ; s is its length


(: area-of-just-squares
    ([Listof Shape] -> REAL))
(define (area-of-just-squares s)
  (define sq* (filter Square? s))
  (define ar* (map sq-area sq*))
  (define sum (foldr + 0 ar*))
  ;; return:
  sum)

(: sq-area (Square -> REAL))
(define (sq-area x)
 (sqr (Square-side x)))
```

Figure 69: Set-based reasoning generalizes to higher-order idioms

the sloppy type definitions and signatures (in comments) of the untyped variant. For example, the first data definition (comment) refers to Circle, an undefined name. A future reader is presumably supposed to conclude that the structure definition of circle introduces this type. During the conversion of such a module into a typed one, a programmer must write down definitions that properly refer to each other, which explains the change in spelling of Circle and Square.

When it comes to the function definition of area-of-just-squares itself, the type checker again validates the existing code against the associated type without ado. It mimics the reasoning concerning the framed application of filter. Typed Racket's type for filter says "if the given predicate represent a subset S of the given list elements, then the result is a list whose elements belong to S." Here, Square? is a predicate that does describe a subset of Shape, namely the left half of the union, and therefore the resulting list contains nothing but instances of Square.

Racket borrows object-oriented programming idioms from JavaScript and Python:

```
#lang racket

(provide add-search)

#;
(Pos = (Cons Natural Natural))
#;
(Opt = (Pos or #false))
#;
(Opt* = ([Listof Opt] or #false))

#;
((Class
   [the-text (-> String)])
  ->
 (Class
   [the-text (-> String)]
   [search   (-> String Opt*)]))


(define (add-search cls%)
  (class cls%
    (super-new)

    (inherit the-text)

    #; (String -> Opt*)
    (define/public (search s)
      (define txt (the-text))
      (regexp-match-positions
        s txt))))
```

```
#lang typed/racket ;; add-mixin.rkt

(provide add-search)

(define-type Pos
 (Pairof Index Index))
(define-type Opt
  (U Pos #false))
(define-type Opt*
  (U (Listof Opt) #false))

(: add-search
   (All (a)
    ((Class #:row-var a
      [the-text (-> String)])
     ->
     (Class #:row-var a
      [the-text (-> String)]
      [search (-> String Opt*)]))))

(define (add-search cls%)
  (class cls%
    (super-new)

    (inherit the-text)

    (: search (String -> Opt*))
    (define/public (search s)
      (define txt (the-text))
      (regexp-match-positions
        s txt))))
```

Figure 70: Object-oriented idioms need more than set-based reasoning

- Classes serve as factories for objects.
- A class may extend another class, but determining which class it extends is a semantic decision not a static one.
- A function may consume or return a class. Racketeers call such functions *mixins*.

In other words, classes are first-class values in Racket.

By convention, names ending in % refer to classes.

Figure 70 presents a module that exports a (simplified version of a) widely used mixin. The purpose of the mixin is to consume a class that allows users to manipulate text, e.g., a message box or a program editor. As documented via its signature, it makes one assumption about its argument, namely that it comes with a method called the-text and that this method returns a string. The mixin extends the given class that supports all methods of the given one, plus a method that searches for a pattern in the string extracted with the-text. More precisely, the new method returns the positions where the pattern occurs in the string.

Writing down the mixin's signature takes three auxiliary data definitions: one for positions (Pos) within strings; another one for optional positions, in case the search for a pattern fails; and a final one for an optional list of search results. The signature itself tells a future reader of this code that the given class must have (at least) a method named the-text and that its result still has the method plus a new one named search.

As the right column of figure 70 indicates, the signature is focus of this example. The framed type signature looks much more complex than the informal one on the left. Concretely it differs in two ways, both major:

- First, the All quantifier signals that the signature is parametric in some parts, and a reader must pay attention to where these parts are.
- Second, the quantified type variable, a, denotes a *row*, that is, any feature that a class type may mention: fields, methods, and more.

Put differently, turning the informal signature into a technically correct takes both an understanding of a sophisticated notation and of universal row polymorphism. Given this type, the type checker can validate an implicit point of the untyped code: the mixin cannot rely on, or modify the meaning of, anything but the the-text method. While the creator of the untyped code intended to issue this rather strong guarantee, Typed Racket can statically validate it for the typed set of modules and can generate runtime checks that guarantee the integrity of this statement at run time.

The presented example are illustrative but far from complete. Once language creators have an initial design of a migratory type system, developers will ask for two kinds of refinements. First, some will argue that it requires too many changes to untyped code to appease the type checker or that it is impossible to appease it all but should be possible. Second, developers used to static type systems and their pragmatic advantages may expect to find certain conventional type-system features that are incompatible with a migratory system. As a result, the development of a migratory type system is a never-ending refinement loop.

2.2 Integrating Typed with Untyped Code

Equipped with a sense of what it is like to program in Typed Racket, we can now address the issue of mixing typed and untyped code. Since Racket modules specify the language of their content, the creators of Typed Racket made the decision to "migrate" at the granularity of entire modules. This decision implies that there are four kinds of module dependencies: the original untyped-on-untyped plus the new typed-on-untyped, typed-on-typed, and untyped-on-typed cases.

Of these four cases, only one poses a design problem to the type-system creators, namely, the case of importing an untyped module into a typed one. Since the type checker needs to know types for all names and since untyped modules do not supply types for their export, Typed Racket supplies one novel construct: require/typed, which, roughly speaking, enables programmers to attach types to imported names.

Next, importing a typed module into another typed module is straightforward. The first module attaches a type to each exported name, and the type checker can use this type for the validation of the second one. Conversely, since the type checker does not inspect untyped modules, exporting from a typed to an untyped merely means ignoring the types from the exports in the importing one.

Let's consider some concrete examples. Suppose a Typed Racket developer needs to create code that calls a factorial function. Further assume that in the decades prior to the creation of Typed Racket, the Racket team equipped its language with a discrete mathematics library that exports !, the factorial function. Here is how the developer would go about importing this one function from the untyped Racket library:

```
# lang typed/racket

(require/typed library/discrete-mathematics ;; imagined
  [! (-> Natural Natural)])

... (! 3) ...
```

This require/typed specification still refers to a specific module, but it also lists all names that are to be imported into this module's scope and, critically, their type. In this example, the type of ! is the same as the one in the preceding section. Note, though, that a type of, say, Natural –> Integer might also work for this client module. That is, the type attached to imported names is up to the developer of client code, not the developers who created the library.

Things can get complicated, though, as the two columns of figure 71 show. On the left, this figure displays a simple data representation for squares and circles, like the one in the preceding subsection. Both structures are exported wholesale, meaning the respective predicates, constructors, selectors, and so on. On the right, the figure displays a typed client that recreates the area-of-just-squares function from figure 69.

```
#lang racket ; server.rkt

(provide
 (struct-out square)
 (struct-out circle))

#; (Shape = Circle or Square)

(struct circle (radius))
#; (circle r) ; r, the radius,
;; is REAL

(struct square (side))
#; (square s) ; s, REAL, is
;; the length of its side
```

```
#lang typed/racket ; client

(require/typed "server.rkt"
  [#:opaque Square square?]
  [#:opaque Circle circle?]
  [square-side (Square -> REAL)])

(define-type Shape
  (U Square Circle))

(: area-of-just-squares
    ([Listof Shape] -> REAL))
(define (area-of-just-squares s)
  ...)

(: sq-area (Square -> REAL))
(define (sq-area x)
  ...)
```

Figure 71: Requiring structures from an untyped module

A sophisticated require/typed specification bridges the gap between the two modules; see framed code. The first two imports name opaque types, Square and Circle, respectively. Each such opaque type corresponds to a predicate, here square? and circle?, respectively. The last import looks like the one from the first example, bringing the square-side selector into scope at type Square –> REAL. By connecting the untyped and typed module with this require/typed specification, the developer can make the rest of the typed module look just like the code in figure 71 (right column).

The last example brings us back to the world of modules and classes from the model in the preceding chapter. Figure 72 displays two modules: an untyped in the left column, a typed one in the right column. While the first one sketches a simplistic multi-line text editor, the second combines this text editor with the mixin from figure 70.

For the purpose of this chapter, the framed require/typed specification is the relevant part of this example. It brings the text-editor% class into scope of the typed module. Since the exporting module, on the left of the figure, is untyped, the importing module must assign a type to the class so that the type checker can validate the module's code. The type specifies the

```
#lang racket ; editor.rkt

(provide text-editor%)

(define text-editor%
 (class object%
   (super-new)

   (field
    #; [Vectorof String]
    (text (make-vector LINES "")))

   #; Natural String -> Void
   ;; add `c` to the end of
   ;; the `at` line `this` editor
   (define/public (text++ at c)
     (define a (vector-ref text at))
     (define b (string-append a c))
     (vector-set! text at b))

   #; -> String
   (define/public (retrieve-all-text)
     (string-join (vector->list text) " "))))
```

```
#lang typed/racket

(provide searchable-text-editor%)

(require add-mixin)

(require/typed editor
 [text-editor%
  (Class
   [field (text (Vectorof String))]
   [text++ (-> Natural String Any)]
   [retrieve-all-text
     (-> String)])])

(define searchable-text-editor%
  (add-search text-editor%))
```

Figure 72: Requiring a class from an untyped module

only field of text-editor% as a string-vector, and associates the two methods—text++ and retrieve-all-text—the expected signatures.

By contrast, the import add-mixin refers to a typed module. Hence it is unnecessary to use require/typed or to equip the imported name with a type. Instead, Typed Racket can retrieve the type of add-mixin from its exporting module. With both the editor and the mixin in scope, the module can create, and export, a searchable version of this text editor by applying the latter to the former.

Stop and take a second look at figure 72. This example serves as inspiration for the model of the next section. Do not continue until you get a good sense of how this example works.

3 Project Language: *Mixed*

Mixed is a model of modern languages with migratory type systems, such as Flow, Hack, and TypeScript. Like all of our models, it deviates from reality in some ways, and it resembles it in others. In order to build on the models from the first eight chapters, the migratory type system of *Mixed* exists at the module level, not inside of small pieces of code. Furthermore, instead of forcing *Mixed* programmers to write “interface modules,” like those found on TypeScript’s DefinitelyTyped repository, the model borrows the idea of require/typed. Although the result is seemingly simple, it still provide the opportunity to investigate the major semantic design decision that creators of these language face.

The section presents *Mixed* following the outlines of the preceding sections: the grammar, the validity constraints, the additional judgment derivation rule, and a note on semantics. When language creators proceed systematically, they approach a design in a similar fashion.

3.1 *Mixed*: the Grammar

Figure 73 presents the essential productions of the BNF grammar of *Mixed*; for all other productions, consult figures 46 and 47. A mixed system consists of typed and untyped modules. The typed ones may import names from the untyped ones and vice versa. In order to facilitate implementations of the model, the new keyword timport marks how a typed module accesses the exports of an untyped module. As mentioned, we borrow the idea of tmodule from Typed Racket, meaning it includes a Shape that specifies the type that the type checker must use to validate the importing module.

```
MixedSystem ::= (MixedModule*
                 MixedImport*
                 Declaration*
                 Statement*
                 Expression)

MixedModule ::= (tmodule ModuleName MixedImport* Class Shape)
              | (module ModuleName Import* Class)

MixedImport ::= (import ModuleName)
              | (timport ModuleName Shape)
```

Figure 73: *Mixed*: the Grammar

It is noteworthy how we can enforce certain contextual constraints via this BNF and the use of distinguishing keywords. First, MixedImport shows up only in the context of tmodule, i.e. an untyped module cannot impose a type on an imported class. Second, the imports for the system’s body also belong to MixedImport, which implies that it is typed code. While this kind of "grammar hacking" accomplishes some basic validity checking, it cannot eliminate the context-sensitive validity checker in general.

Exercise 78. Design an AST data representation for *Mixed*. Implement a parser for *Mixed* that maps an S-expression to an instance of AST. Re-use your solution from exercise 51 as much as possible.

3.2 *Mixed*: Validity

Even though the BNF forces programmer to use timport only in the context of tmodule, this constraint does not ensure that the given ModuleName refers to an untyped module. Similarly, the grammar does not rule out the attempt to import a typed module via timport. Finally, a sequence of MixedImports could contain two distinct timport specifications for the same untyped module. In principle, this situation is acceptable but, to simplify the model implementations, we rule it out.

Technically put, a *Mixed* system is subject to three structural validity constraints:

- The typed portions of a system must import untyped modules via a timport specification.
- The typed portions of a system may not import the same module using two different Shapes.
- The typed portions of a system may not import a tmodule using a timport specification.

Keep in mind that "typed portion" covers typed modules and the system's body consisting of declarations, statements, and the final expression.

Exercise 79. Design and implement a validity checker that enforces the validity rules for *Mixed*. The checker consumes error-free ASTs from exercise 78; if it finds errors it annotates the AST appropriately.

3.3 *Mixed*: Type Checking

The point of creating mixed systems is to connect typed and untyped pieces of code; in the context of *Mixed*, this means connecting typed and untyped modules. Hence, a type checker must validate that the types and the code in the typed portions are in sync. Conversely, the untyped modules of the same system do not have to satisfy any type judgment rules.

$$\frac{\begin{array}{l}\text{M is the name of mod in Modules}\\ \text{C is the name of the class defined in mod}\end{array}}{\text{Modules, SClasses} \vdash \text{(timport M S)} \Longrightarrow \text{SClasses [C : S]}} \quad \text{[an import]}$$

Figure 74: Deriving the key judgment for mixed systems

Given this objective, the adaptation of the type judgment rules from chapter VIII is straightforward:

- The adaptation of the rule for type checking complete systems, see figure 51, must ignore untyped modules.
- The import of an untyped module into a typed one calls for the additional type judgment rule of figure 74.

In sum, and as advertised at the beginning of this chapter, the *Mixed* model is a rather small extension of *Module* from a syntactic point of view. By implication, the type system of *Mixed* does not accommodate the programming idioms of *Module*; it imposes the same limitations on typed portions as the model of the preceding chapter. The goal remains to investigate the semantic choices in the design of *Mixed*.

```
(module A             (tmodule clientA    (tmodule clientB
 (class A ()           (timport A          (timport A
   (method id(x)         ( ()                ( ()
     x)))                  ((id (REAL)         ((id ( (((x REAL)) () ))
                               REAL))))            ( ((x REAL)) () )))))
                        ...)               ...)
```

Figure 75: Two types for the same exported class

Exercise 80. Figure 75 consists of three columns:

- The left-most one displays the untyped module A, which exports a class named A.
- The middle column shows how to import module A at one type.

- The module in right-most column imports module A at a different type.

Argue why the type checker should accept both timports or reject them.

Exercise 81. Adapt the type checker from exercise 62 to perform a complete type-checking pass for *Mixed*. The type checker should merely signal an exception when it discovers a type violation. Each such exception should come with an error messages that assist the *Mixed* programmer with the task of repairing type errors.

3.4 A Note on Types and Semantics

According to exercise 80, two distinct modules can legally import the same module with two distinct types. Similarly, a module can import another module using a type that the type checker would reject. Figure 76 illustrates this point with the same exporting module as figure 75, but with an importing module that assigns a invalid type. That is, the type checker would reject this combination of exported class with importing type.

```
(module A
  (class A
    (method id(x)
      x)))
```

```
(tmodule clientA
 (timport A
  ( ()
    ((id (REAL)
       ( ((x REAL)) () )))) )

 (class clientA ()
   (method di() (new A())))

 ( ()
   ((di ()
     ( ()
     ((id (REAL)
       ( ((x REAL)) () ))))))))
```

Figure 76: The wrong type for an exported class

Stop! Convince yourself that the type checker rejects a typed variant of module A if it is equipped with the framed type inside of clientA's import.

Exercise 82. The following code fragment completes the two modules of figure 76 into a full *Mixed* system:

```
    (import clientA)

    (def a (new A()))
    (a --> di())
```

Does this system type check? If so, explain how. Otherwise explain why the type checker rejects this system.

The exercise raises the question of what should happen with a *Mixed* system that type checks even though the type assigned to a class imported from an untyped module is a mismatch. As alluded to in this chapter already, this question leads to a semantic design decision, and as it turns out, different language creators give different answers to this question.

4 Semantics: The TypeScript Approach

TypeScript advertises itself as a form of JavaScript. Concretely, its web page says "TypeScript becomes JavaScript via the delete key." What this means, is straightforward. once the type checker has confirmed the validity of the types with respect to the typed portions of the code, the TypeScript implementation traverses the given code, erases the types, and generates a plain JavaScript program.

Modeling TypeScript's semantics with our models is equally straightforward. Every tmodule of a *Mixed* system becomes a plain module; the type of the former simply disappears. Similarly, every timport specification becomes a import by erasing the type from the former. The resulting code is a well-formed and valid an element of the *Module* model.

Exercise 83. Your task is to implement the "erasure" pass, which turns a *Mixed* system into a *Module* system.

5 Semantics: The Typed Racket Approach

In contrast to the industrial TypeScript team, academic researchers have spent two decades exploring different flavors of sound semantics of migratory type systems. The goal of these efforts is to notice when a value flows from an untyped portion of the code into the typed portion, or vice versa, but does not match the specified type.

For example, if the di method in figure 76 not only created an instance of A but also called the instance's id method on some number, id would return the very same number, which does not match its return type specified in the framed timport type. If such a value flow takes place in a TypeScript program compiled to JavaScript, it is unclear what will happen; academic language designs aim to catch such *type mismatches* and inform the software developer of them.

Typed Racket was the first language to implement a complete solution.

The stated goal allows many different implementations. Each implementation notices type mistakes at slightly different times. This section presents an implementation that approximates Typed Racket's solution, but deviates from it in some ways. In essence, the Typed Racket solution is

- to check basic values, such as numbers or strings, when they flow from an untyped to a typed portion of the code or vice versa; and

- to wrap objects so that it becomes possible to check every access or mutation of the value according to the specified types.

For simplicity, the semantic model of this section leverages the existence of the linker to manifest the boundaries between typed and untyped portions. Concretely, the linker is used to clone an untyped module if it is subject to a timport in the context of typed code. This clone comes with a type but is *not* type checked; after all, type checking takes place before the linker kicks in. Finally, the section presents a modified CESK machine, which creates simple proxy values when a class from a tmodule is instantiated. This proxy value monitors all interactions between the object and its use contexts, and when a type mismatch is discovered, it causes the machine to halt execution.

5.1 Linking in Mixed-Type Setting

The stated semantic goal of wrapping all instances of typed classes in a protective cocoon poses a problem. While the class definitions inside of typed modules obviously give rise to typed instances, the import of untyped classes into typed portions of a *Mixed* system implicitly creates typed classes. After all, the type checker validates the type portion as if the untyped module were type-checked, too. If the semantics is supposed to ensure that all instances of typed classes behave, and are used, in accordance with their type specification, it must be informed of this implicit conversion of an untyped module into a typed one.

At this point, you might wonder why we cannot just use the types in timports to equip the untyped modules with types. Doing so would convert them into typed modules, and the semantics could generate the desired wrapper. It turns out, though, that this idea really is too simplistic. As figure 75 demonstrates, the two distinct typed modules can import the same untyped module at two distinct types without violating validity or type constraints.

Instead of imposing additional validity constraints, we make the additional linker pass work harder. To differentiate this new linker from the one in chapter VII, we call it a *sound linker*. Its purpose is to translate a well-formed and valid *Mixed* system into a well-formed and valid *Classy* program, using the old linker pass and two new ones:

1. For every occurrence of a timport specification, a *synthesis* pass generates a new typed module from the named untyped one:

   - For an occurrence of (timport M Shape) in module K, this pass creates a typed copy of M named M.into.K annotated with Shape and replaces the import specification in K with (import M.into.K).
   - For an occurrence of (timport M Shape) in the system's body, this pass creates a typed copy of M named M.into.Body annotated with Shape and replaces the import specification in the system's body with (import M.into.Body).

   Note These names make sense only because this synthesis pass may assume the AST is well-formed and valid. It would be a problem if the system contained an untyped module named Body. So, *do not use Body as a ModuleName.*

2. A *typed-classes* pass transfers the types from tmodule ASTs to the ASTs of class definitions.

   The modified CESK machine of section 5.2 will use these type annotations in class ASTs to monitor all of their instances.

Exercise 84. Design and implement the *synthesis* pass so that it adds the generated modules to the given *Mixed* system according to the specifications in this section. Keep in mind that the given AST is well-formed, valid, and type-checked.

Exercise 85. Design and implement the *typed-classes* pass according to the specifications in this section. It consumes the output of the *synthesis* pass from exercise 84.

Exercise 86. Adapt the linker from exercise 64 so that it works for the ASTs generated by the *typed-classes* pass exercise 85.

5.2 A CESK Machine Based on Proxies

A sound semantics for *Mixed* monitors instances of typed classes in the *Classy* program that the linker produces, Monitoring means inspecting values that flow into or out of instances via field access, field modification, or method calls. That is, when some code retrieves the value of field f from an instance of typed class C, then monitoring ensures that the value conforms to the type specification of f in C's type. Similarly, when some code calls method m of an instance of typed class C, the monitor checks whether the given argument values conform to the domain part of m's type signature in the type of C.

Let's enumerate the three ideas that inform the adaptation of the CESK machine from chapter VI to *Mixed*:

1. An object is monitored if it is an instance of a typed class.
2. A monitoring semantics intercepts values as they flow into or out of an object through any channel.
3. An intercepted value must conform to its type specification, meaning the semantic model must come with a *conformance* relation.

The remainder of the section introduces these ideas one at a time.

Note A moment's thought suggests that the second idea is too strict. Consider a method call whose target object is an instance of a type-checked class. The purpose of type checking is to guarantee that the method's code is in sync; in particular, the method returns a value that has the specified type, assuming the given arguments conform to their respective specifications. Hence, it should suffice to intercept only the arguments to such a method and to check their conformance. A similar argument could reduce the number of interceptions for field mutation and reference. While the *Mixed* model uniformly intercepts all values to keep things simple, the Typed Racket implementation reduces the number of interceptions as much as possible.

Your choice of data representation for objects in the CESK machine includes a reference to the class of which it is an instance. Hence, the machine can always determine whether an instance belongs to a typed class, because the *Mixed* linker adds types to the ASTs of classes. However, since the modified CESK machine must retrieve types frequently, it is best to combine objects with types when they are created from a typed class. Following the Typed Racket creators' terminology, we call this combination a *proxy*, and

	Control	Environment	Store	Kontinuation
evaluate a 'new' expression (untyped class, success)				
before:	(new C (x* . . .))	e	s	k
subject to:	class C is untyped and has the same number of fields as in x* ...			
after:	obj	e	s	k
where	v*, . . . = s[e[x*]], . . .			
and	obj = create instance of C using v*, . . . for the values of the fields			
evaluate a 'new' expression (typed class, success)				
before:	(new C (x* . . .))	e	s	k
subject to:	class C is typed and x* ... conforms to its field types			
after:	prx	e	s	k
where	v*, . . . = s[e[x*]], . . .			
and	obj = create instance of C using v*, . . . for the values of the fields			
and	typ = the type of class C			
and	prx = make obj conform to type typ			
evaluate a 'new' expression (failure)				
before:	(new C (x* . . .))	e	s	k
subject to:	class C is typed and ... x* ... fails to conform to its field types ... or is untyped ... and x*... is of incorrect length			
after:	error	[ ]	[ ]	[ ]

Figure 77: The proxy-based CESK machine: creation

we agree that the modified CESK machine considers them as values and monitors their interaction with other code.

Exercise 87. Design a data representation for proxy values, which combine objects with the types of their classes. See exercise 23 for the data representation of objects. Include the functionality for retrieving the types of fields, domain types of methods, and range types of methods.

Figure 77 presents the machine transitions of the modified CESK machine for creating new instances from classes:

1. If the class comes without a type and the new expression supplies the correct number of variables as arguments, the machine retrieves the values of these variables and creates a regular object.

2. If the class is typed, the machine retrieves the values again and, additionally, makes sure that these values conform with the types of the class's fields. Instead of just creating an object, the machine forces the object to conform to the class's type, which in this case means it combines the two into a proxy.

3. Finally, if neither condition holds, the CESK machine signals an error.

Stop! Compare the first two rules so you understand the differences.

Given an object, a program can check whether it is an instance of a particular class. While the top half of figure 78 displays a transition rule that looks just like one from the original CESK specification (see chapter VI), it differs in a subtle but critical manner. And, understanding this difference takes some effort.

	Control	Environment	Store	Kontinuation
evaluate an 'isa' expression				
before:	(o isa C)	e	s	k
after:	ans	e	s	k
where obj = s[e[o]]				
and ans = check whether obj is instance of C				

the modules

```
(module Untyped
 (class U ()
   (method m(o) o)))

 (tmodule Typed

   (timport Untyped
     (()
      ((m (Number)
         Number))))

   (class T ()
     (method m(o) (o isa B)))

   (()
    ((m ((() ((m (Number) Number))))
          Number))))
```

the system body

```
(import Typed)

(timport Untyped
  (()
   ((m (Number)
      Number))))

(def u (new U()))
(def t (new T()))

(t --> m(u))
```

Figure 78: The proxy-based CESK machine: inspection

Take a look at the bottom half of figure 78. It illustrates the subtlety of isa in this setting with a complete sample system. While the left column displays the modules of this *Mixed* system, the right one shows its body. Note how the framed timport specifications bring the untyped class U from module Untyped into the scope of Typed and the system's body respectively—attaching the exact same type.

Stop! What do you expect as the outcome of this system?

Here is the point. Although the system body seems to create an instance of U at the same type as module Typed expects, running the program yields 1.0, meaning that u is *not* considered an instance of U inside of method m in class A. Why?

With the introduction of synthetic modules and proxy instances created from their classes, the semantics also adds error behavior. That is, depending on context, an interaction with an instance of a class from a synthetic module may signal an error due to the superimposed type. In our running example, two distinct import specifications for Untyped can attach two distinct types. Hence, one instance may behave properly when it interacts with a piece of code, while the other would signal an error during an interaction with the exact same piece of code. It is therefore necessary to distinguish instances of two classes from each other, even if the source of these classes is one and the same untyped module.

Exercise 88. Work out the *Classy* program that the (adapted) linker should produce. What would the names of B in Typed and the system's body be, respectively?

Exercise 89. Design the auxiliary function needed to determine whether some potentially proxied object is an instance of some class.

Figure 79 presents the next new cases of the CESK transition function: the successful reference to a field of an object and the revised failure case. When the C register contains a field access expression, the machine also checks whether the value in the named field conforms to the type specified in the proxy. If so, it retrieves the value, enforces conformance, and places the conforming value into the C register. Regardless of the nature of the target, if the field is missing or the value in the field of a proxy does not match the specified type, the CESK machine signals an error.

Stop! Take a close look at the formulation of the precondition in the success case of figure 79 and the last line of the *after* state formulation.

While the precondition says "the value conforms to its type," the *after* state uses the strong language of "making the field value conform to its type." The second formulation indicates that conformance checking is not

	Control	Environment	Store	Kontinuation
evaluate a field reference (proxy)				
before:	(p → f)	e	s	k
subject to:	p is a proxy and the value of the field conforms to its type			
after:	v	e	s	k
where	prx	=	s[e[p]]	
and	[obj, fieldType]	=	extract object and type of fueld f from proxyprx	
and	u	=	get value of f from obj	
and	v	=	make u conform to fieldType	
evaluate a field reference (failure)				
before:	(o → f)	e	s	k
subject to:	o is not an object/proxy with the correct field property			
after:	error	[ ]	[ ]	[ ]
where	obj = s[e[o]]			

Figure 79: The proxy-based CESK machine: field reference

just an up or down vote on the relationship between a value and its specified type.

Indeed, checking whether some given value *conforms* to a type specification has *three* possible outcomes:

1. If the value matches the type, the value is returned.

2. If an object *might* live up to a type specification in the future, the conformance function creates an appropriate proxy from the object and the type.

3. Otherwise, the given value—a number, an object, a proxy—does *not* conform to the given type.

Figure 80 spells out the six scenarios via a two-dimensional table. The horizontal direction enumerates the three kinds of values that the conformance checker might encounter; the vertical one lists the two kinds of types.

Stop! Read the two complex cases in the lower right of the table in figure 80 carefully.

Two of the cases in figure 80 deserve a detailed explanation. When the given value is a proxy and the type is some Shape, the conformance checker must compare types, meaning the CESK machine checks types for equality if the value is:

	a number n	an object obj	a proxy prx
Real	use smaller{n}	**no**	**no**
Shape S	**no**	combine obj and S into a proxy	if the type in prx is equal to S: use prx else: **no**

Figure 80: A value conforms to a type, if ...

at run time. This check is an artifact of the model; Typed Racket does *not* retain types beyond the type-checking step.

When the given value is an object and the type is some Shape, the object may live up to the expectations. But obviously, it is impossible to know at this particular point whether the rest of the computation is going to interact with the object in an appropriate manner. Hence, the machine combines the object and the type into a proxy and continues from here.

The conformance could check some "first order" properties, such as whether the set of method names are the same.

	Control	Environment	Store	Kontinuation
execute a field assignment (proxy)				
before:	u	e	s	《 [ ], (p → f = rhs)::stmt*, r 》
subject to:	p is a proxy and the value of the field conforms to its type			
after:	†	e	s	《 [ ], stmt*, r 》
where	prx	=	s[e[p]]	
and	[obj, fieldT]	=	extract object and type of fueld f from proxyprx	
and	v	=	make u conform to fieldT	
and			set f in obj to v	
execute a field assignment (failure)				
before:	v	e	s	《 [ ], (o → f = rhs)::stmt*, r 》
subject to:	o is not an object/proxy with the correct field property			
after:	error	[ ]	[ ]	[ ]
where	obj = s[e[o]]			

Figure 81: The proxy-based CESK machine: field mutation

Let's study field mutation next. Unlike field reference, which extracts a value from an object, field mutation injects one. According to our agreement, the CESK machine must intercept this value and inspect it, if the target object is proxied. Like for field references, this constraint requires one new case for the transition function and one modified one:

- If the target of a field mutation is a proxy and the value in the C register conforms to the field's type, then the CESK machine assigns a type-conforming value to the corresponding field of the object representation.
- if the field is missing or the value to be stored in the field of a proxy does not match the specified type, the CESK machine signals an error.

See figure 81 for these two cases of the revised transition function.

	Control	Environment	Store	Kontinuation
evaluate a method-call expression (proxy)				
before:	(p → m (x* . . .))	e	s	k
subject to:	p is a proxy and the arguments conform to the method type			
after:	†	e1	s1	push[cl, push[rangeT, k]]
where	prx	=	s[e[p]]	
and	[obj, domainT, rangeT]	=	extract the object and the type of method m from prx	
and	tmp*, . . .	=	s[e[x*]], . . .	
and	arg*, . . .	=	make the tmp*. . . values conform to the domainT types	
and	[para*, body]	=	the parameters & body of method m per the class of obj	
and	cl	=	*closure*: body *in* e	
and	thisL	=	a new (relative to s) location	
and	xl*, ...	=	as many new (relative to s) locations as elements in x*, ...	
and	e1	=	[ ][this = thisL][para* = xl*], . . .	
and	s1	=	s[thisL = obj][xl* = arg*], . . .	
evaluate a method-call expression (failure)				
before:	(o → m (x*))	e	s	k
subject to:	o is not an object/proxy with the correct properties for this call			
after:	error	[ ]	[ ]	[ ]

Figure 82: The proxy-based CESK machine: method calls

A method makes up a two-way street for interactions between an object and its context. On call, a method internalizes values. On return, a method externalizes values. In both cases, these values have to conform to the types specified in a proxy.

Let's consider each direction separately, starting with method calls. The rules are spelled out in figure 82. A method call whose target is a proxied object may proceed if the argument values conform to the domain part of the method's signature. If these conditions hold, the CESK machine retrieves the proxy and extracts from this proxy the object, the domain types, and the range type. At this point it can make the argument values conform to the domain types. Using the resulting values, the CESK proceeds in a manner that is similar to regular method calls.

Stop! Take a close look at the *after* state of this case in figure 82.

Note how the K register of the *after* state contains a continuation that comes with *two* new frames. The top-most one is the usual closure. But the second from the top is just a type. It is at this point in the evaluation of a method call that the CESK machine knows to which return type a potential return value must conform, and it must store this type in the continuation so that conformance can be checked upon method return. And this brings us to the final two rules.

Exercise 90. Explain the second case in figure 82.

	Control	Environment	Store	Kontinuation
returning from a proxied method call (success)				
before:	v	e	s	(type: rangeT)::k
subject to:	value v conforms to rangeT			
after:	rv	e	s	k
where	rv = make v conform to type rangeT			
returning from a proxied method call (failure)				
before:	v	e	s	(type: rangeT)::k
subject to:	value v does not conform to rangeT			
after:	error	[ ]	[ ]	[ ]

Figure 83: The proxy-based CESK machine: method returns

A method return takes place when the C register contains a value and the top of the K register is a type. It is exclusively due to a call that targets a proxied object. Figure 83 displays the two corresponding cases of the revised CESK transition function:

- The first case deals with a return value v that conforms to the specified type. If so, the CESK machine forces v to conform and places the resulting value rv into C all while popping the continuation in K.

- The second case concerns failure. If the return value fails to conform to the specified type, the CESK machine signals a failure.

The introduction of a return-from-method case into the transition function has a serious consequence. As section 4.1 in chapter VI demonstrates, a method that calls itself consumes as much continuation space as a while loop. For the proxy-based CESK machine, this property no longer holds. A self-calling method from a proxied object is going to grow the continuation one frame per call, meaning in real-world implementations such methods may run out of stack space.

Exercise 91. Besides the new and revised transition rules presented in this section, the proxy-based CESK machine also needs a revised function for checking the structural equality of values. After all, the addition of proxies to the set of values clearly calls for a revision.

Revise your function for determining the structural equality of values so that it unwraps the objects hidden inside of proxies. See exercise 25.

Exercise 92. Revise the data representation from exercise 24 to accommodate proxies.

Exercise 93. When the proxy-based CESK machine encounters objects, they are instances of untyped classes. Hence interactions with these objects may go wrong just like they did in chapter VI.

Restore the run-time checks to the CESK machine that you eliminated in response to exercise 66.

Exercise 94. Revise and extend the transition function from exercise 33 so that it properly works with proxies.

By combining the existing load, unload and transition functions with the runMachine function, you obtain a complete semantics for well-formed and valid *Mixed* programs.

Exercise 95. Come up with an example system in *Mixed* that creates instances of type-checked and classes from untyped modules. Make sure these instances interact with each other.

6 The Very Final Bit of Theory: Real Type Soundness

Section 4.1 of chapter VIII explains which run-time checks in the CESK transition function are superfluous *if* the relationship between the type checker and the transition function satisfies a type soundness theorem. This chapter raises the question of whether such a theorem can hold for *Mixed* or the actual language implementations that *Mixed* models.

The answer mat come as a bit of a surprise:

> *Language models that mix typed and untyped code are type sound to a degree, not in an absolute sense.*

Put differently, "type soundness" theorems for modules like *Mixed* form a spectrum. Depending on what kind of conformance checks they impose, they catch some or all type mismatches and, if they catch them, they catch them at different times during program evaluation. The very fact that this chapter presents *two* semantics for *Mixed* is proof of this claim. While the semantics of section 4 checks *none* of the interactions between typed and untyped code, the semantics of section 46 checks *all* interactions.

One consequence of this lack of complete type soundness concerns the run-time checks in CESK machines. They cannot be removed, which is why exercise 93 has you restore those removed in the preceding chapter. Another consequence is about pragmatics. Run-time checks exist to catch type mismatches, which programmers introduce accidentally. And this brings us to the pragmatics of *Mixed* and its stark semantic choices.

They also protect partial primitives such as division. We ignore those.

Note On reflection, you may realize that real language implementations are closer to the *Mixed* model than the models of the preceding chapters. That is, all languages mix type-checked and unchecked code. For example, the latter come in the form of run-time libraries that access devices, and those libraries tend to use languages with few protective features. Hence, "type soundness to a degree" for their models is the most realistic characterization a language researcher can provide—even if we ignore the distinction between models and implementations with bugs.

7 Pragmatics: Migratory Type Systems

Languages with migratory type systems offer different answers to semantics-oriented pragmatics issues than languages with plain static type systems. As the two sections on semantics point out, an implementation can choose from two radically different options:

- to erase the types and just run the resulting untyped code; or
- to use run-time checks to enforce types at the boundaries between typed and untyped code.

The choice clearly has major implications for the work of software developers. Concretely, the corresponding questions are why an implementation team would choose

- to erase the types and thus fail to inform developers about type mismatches, or
- to use run-time checks and to impose the cost of these checks on the code that developers deploy.

Stop! Take a look at figure 84 and identify the work situations to which these questions alude.

work situation	a programming with a migratory type system	
	with type enforcement	with type erasure
design code		
...	...	...
— migrate		
testing		
debugging		
deployed code		
— bug diagnosis		
...	...	....

Figure 84: Work situations and the role of types systems

7.1 The Twp Semantics with Concrete Examples

Before we discuss answers to these two questions, let's consider an illustrative and comparable example from two real-world implementations of *Mixed*: TypeScript combined with JavaScript, and Typed Racket combined with Racket. As mentioned, their type systems resemble each other but their semantics roughly correspond to the two sections on semantics, respectively.

Figures 85 and 86 present the same program in the two combinations. Both figures display two columns, with one file each. The left column is a *typed* module for managing (simplistic) bank accounts; the right one shows a naive client module. Each of these client modules interacts with the typed module in two ways:

- The first function call to deposit requests a type-correct addition of 100 to the current balance.
- The second function call applies deposit to " pennies!", a string. Obviously this application represents a type mismatch, because deposit expects a number or Natural, respectively.

bank.ts

```
// balance in $$s
var blnc = 0;

// add `amt` in $$s to `blnc`
export
 function deposit(amt: number) {
  blnc += amt;
}

export
 function printBlnc() {
  console.log("balance: $" + blnc);
}
```

client.js

```
var Bank = require('./Bank.js');

// correct interaction
Bank.deposit(100);
Bank.printBlnc();

// incorrect interaction
Bank.deposit(" pennies!");
Bank.printBlnc();
```

Figure 85: A TypeScript/JavaScript type mismatch

bank.rkt

```
#typed/racket
(provide deposit printBlnc)

;; balance in $$s
(define blnc : Integer 0)

;; add `amt` to $$s to `blnc`
(define (deposit amt : Natural)
  (set! blnc (+ blnc amt)))

(define (printBlnc)
  (println (~a "balance: " blnc)))
```

client.rkt

```
#lang racket
(require "bank.rkt")

;; correct interaction
(deposit 100)
(printBlnc)

;; incorrect interaction
(deposit " pennies!")
(printBlnc)
```

Figure 86: A Typed Racket/Racket type mismatch

Stop! Inspect the two figures and make sure you understand the code and the two calls in the client module before reading on.

Following the presentation of the two semantics, the two code combinations behave differently. While the TypeScript/JavaScript combination prints

```
balance: $100

balance: $100 pennies!
```

to the console, the Typed Racket/Racket system signals an error after printing the balance in response the first function call:

```
balance: $100
deposit: contract violation
expected: natural?
given: " pennies!"
in: the 1st argument of
    (-> natural? any)
contract from: bank.rkt
blaming: client.rkt
 (assuming the contract is correct)
```

The error message deserves some explanation. It informs the developer about the type mismatch as the failure of ″pennies!″ to satisfy natural?, the predicate that enforces the type Natural. Furthermore, it explains which part of the function signature the value violates and which boundary between typed and untyped code causes the problem. Do note the last line, which is a warning to the developer and which is important when it comes to the pragmatics rationale behind the semantic choice of enforcing types.

Note In general, a mixed TypeScript/JavaScript program does not have to terminate properly when a type mismatch occurs. Like our CESK model, the JavaScript machine does perform some checks that protect primitive computational operations. Hence these checks may *eventually* catch a type mismatch because some inappropriate value flows into a run-time check.

As the example points out, it performs fewer checks and thus fails to "protect" addition from strings.

7.2 Performance

Performance matters. And every run-time check has the potential to affect a program's performance.

After type checking, the TypeScript implementation compiles the given source code to plain JavaScript, mostly by stripping the types. Since software developers have experienced tremendous performance gains over the past decades, they consider JavaScript as a performant language. If the TypeScript implementation were to add run-time checks, a developer's work might suffer in two ways:

- First, the translated TypeScript code would have to pay for the run-time checks during execution. The developer would have to measure whether the cost of these run-time checks is acceptable and, if not, conduct a performance-debugging session.

- Second, as is, TypeScript code carries performance "on its sleeves." An experienced JavaScript developer can "see through" the TypeScript source code and understand it in terms of the corresponding JavaScript code, including its performance. If the compilation injected run-time checks, this direct correspondence would be severely distorted.

Plus readers who remember section 4 in chapter VIII

Experienced language creators may object to this analysis. They understand that type-checking plus a soundness theorem enable language implementers to eliminate run-time checks that ensure the proper working of computational primitives. Conversely, they may wonder why type checking the typed portions of a mixed system does not yield similar benefits.

In the case of JavaScript, the answer is that the implementation observes the kind of values that flow through a program, recognizes where it can eliminate the checks that a static type system would eliminate, and recompiles the code accordingly. Additionally, since misbehaving untyped values may show up inside of TypeScript portions of mixed systems, the translation is not sound and therefore cannot rely on type checking.

In the case of Typed Racket, the answer differs. The Typed Racket implementation *can* rely on the type checker because run-time check guarantees that the validated logical properties hold during program execution. As a result, programs written in *just* Typed Racket do benefit from type checking just like programs written in a statically typed language. Even mixed systems written in both Racket and Typed Racket exhibit improved performance on occasion.

In some situations, however, Racket/Typed Racket combinations exhibit steep performance losses. The creators of the language have reported slow-downs at the level of many orders of magnitude for object-oriented mixed systems, while mostly-functional code seems to suffer from a mostly reasonable cost for the run-time checks. At this point in time, it is not clear whether these costs are intrinsic to languages with somewhat-sound migratory type systems or whether different implementations of the underlying untyped language can eliminate most of the cost of run-time checks.

7.3 Testing, Debugging, and Deployment

Discovering type mismatches matters. And every failure to discover a type mismatch or every delay in the discovery process affects the working software developer's in several different situations, including testing, debugging, and system deployment.

As section 7.1 demonstrates, mixed TypeScript/JavaScript systems such as the one from figure 85 do not discover the type mismatch between the string " pennies!" and the type number. The result is an unexpected outcome. If this example were a test case, the developer would add an expected outcome and an automatic comparison, and the test-case failure would point out *a* problem.

What just this simple example shows then are three obvious consequences for the developer's work:

- The lack of run-time tests shifts work from the language creator to the software developer. It becomes the task of the latter to write tests that catch type mismatches.

- A failing test case provides less information than the failure of a run-time check that identifies the boundary that causes the type mismatch and the value that causes it. It is now the developer's problem to find this boundary and ideally to determine which value failed to match the type.

- Finally, a type mismatch may also affect a system's behavior after it is deployed. In this case, it is the user who suffers. And, if this user reports the problem, the developers assigned to this bug have even less information than a failing test case. As a matter of fact, they will first have to develop a test case that reproduces the problem and use the failure of this test case to start the search for the type-mismatch error.

Stop! Identify the cells in the table of figure 84 that match the three consequences. Can you think of additional consequences?

By contrast, the Typed Racket/Racket system from figure 86 signals an error as soon as it discovers the type mismatch and supplies a good amount of error information. At first glance, this information should assist a developer with testing, debugging, and even with failures in deployed software. Research by the authors suggests, however, that these appearances are a bit deceiving.

When a type does not match a value, it is possible that the type is wrong, the value is wrong, or both are wrong. And as it turns out, the information in the error messages seems to help in different ways, depending on what is the case:

- If it turns out that the type signature is mistaken, the information from Typed Racket's failing run-time checks seem to help developers quite a bit with the debugging process. As mentioned already, problems with *post hoc* type signatures are common in TypeScript's *Definitely Typed* code repository. So this research result points in a helpful direction.
- If the problem is due to a bug in the code that is equipped with types on a *post hoc* basis, the information seems less helpful. More precisely, the safety checks of Racket's version of a CESK machine already catch many type mismatch problems, and the information that they supply appears to be equally helpful.

In short, the pragmatics of testing and debugging in a language with a migratory type systems poses interesting research problems, and the outcome is wide open.