# The Typed Racket Guide

Version 9.4.0.3

Sam Tobin-Hochstadt <samth@racket-lang.org>,
Vincent St-Amour <stamourv@racket-lang.org>,
Eric Dobson <endobson@racket-lang.org>,
and Asumu Takikawa <asumu@racket-lang.org>

October 28, 2026

Typed Racket is Racket's gradually-typed sister language which allows the incremental addition of statically-checked type annotations. This guide is intended for programmers familiar with Racket. For an introduction to Racket, see *The Racket Guide*.

For the precise details, also see *The Typed Racket Reference*.

# 1 Quick Start

Given a module written in the `racket` language, using Typed Racket requires the following steps:

1. Change the language to `typed/racket`.
2. Change the uses of `(require mod)` to `(require typed/mod)`.
3. Annotate structure definitions and top-level definitions with their types.

Then, when the program is run, it will automatically be typechecked before any execution, and any type errors will be reported. If there are any type errors, the program will not run.

Here is an example program, written in the `racket` language:

```
#lang racket
(struct pt (x y))

; distance : pt pt -> real
(define (distance p1 p2)
  (sqrt (+ (sqr (- (pt-x p2) (pt-x p1)))
           (sqr (- (pt-y p2) (pt-y p1))))))
```

Here is the same program, in `typed/racket`:

```
#lang typed/racket
(struct pt ([x : Real] [y : Real]))

(: distance (-> pt pt Real))
(define (distance p1 p2)
  (sqrt (+ (sqr (- (pt-x p2) (pt-x p1)))
           (sqr (- (pt-y p2) (pt-y p1))))))
```

## 1.1 Using Typed Racket from the Racket REPL

It is possible to use Typed Racket from the Racket REPL. To do so, start Racket with the following command line:

```
racket -I typed/racket
```

# 2 Beginning Typed Racket

Recall the typed module from §1 “Quick Start”:

```
#lang typed/racket
(struct pt ([x : Real] [y : Real]))

(: distance (-> pt pt Real))
(define (distance p1 p2)
  (sqrt (+ (sqr (- (pt-x p2) (pt-x p1)))
           (sqr (- (pt-y p2) (pt-y p1))))))
```

Let us consider each element of this program in turn.

```
#lang typed/racket
```

This specifies that the module is written in the `typed/racket` language, which is a typed version of the `racket` language. Typed versions of other languages are provided as well; for example, the `typed/racket/base` language corresponds to `racket/base`.

```
(struct pt ([x : Real] [y : Real]))
```

This defines a new structure, named `pt`, with two fields, `x` and `y`. Both fields are specified to have the type `Real`, which corresponds to the real numbers. The `struct` form corresponds to its untyped counterpart from `racket`—when porting a program from `racket` to `typed/racket`, simply add type annotations to existing field declarations.

Typed Racket provides modified versions of core Racket forms, which permit type annotations. Previous versions of Typed Racket provided these with a `:` suffix, but these are now only included as legacy forms for backwards compatibility.

```
(: distance (-> pt pt Real))
```

This declares that `distance` has the type `(-> pt pt Real)`.

The type `(-> pt pt Real)` is a function type, that is, the type of a procedure. The input type, or domain, is two arguments of type `pt`, which refers to an instance of the `pt` structure. The `->` indicates that this is a function type. The range type, or output type, is the last element in the function type, in this case `Real`.

If you are familiar with contracts, the notation for function types is similar to function contract combinators.

```
(define (distance p1 p2)
  (sqrt (+ (sqr (- (pt-x p2) (pt-x p1)))
           (sqr (- (pt-y p2) (pt-y p1))))))
```

This definition is unchanged from the untyped version of the code. The goal of Typed Racket is to allow almost all definitions to be typechecked without change. The typechecker verifies

that the body of the function has the type `Real`, under the assumption that `p1` and `p2` have type `pt`, taking these types from the earlier type declaration. Since the body does have this type, the program is accepted.

In the Typed Racket REPL, calling `distance` will show the result as usual and will also print the result's type:

```
> (distance (pt 0 0) (pt 3.1415 2.7172))
- : Real
4.153576541969583
```

Just evaluating the function name will print the function value and its type, which can be useful for discovering the types that Typed Racket ascribes to Racket functions. Alternatively, the `:print-type` command will just print the type:

```
> distance
- : (-> pt pt Real)
#<procedure:distance>
> string-length
- : (-> String Index)
#<procedure:string-length>
> (:print-type string-ref)
(-> String Integer Char)
```

## 2.1 Datatypes and Unions

Many data structures involve multiple variants. In Typed Racket, we represent these using *union types*, written `(U t1 t2 ...)`.

```
#lang typed/racket
(define-type Tree (U leaf node))
(struct leaf ([val : Number]))
(struct node ([left : Tree] [right : Tree]))

(: tree-height (-> Tree Integer))
(define (tree-height t)
  (cond [(leaf? t) 1]
        [else (max (+ 1 (tree-height (node-left t)))
                   (+ 1 (tree-height (node-right t))))]))

(: tree-sum (-> Tree Number))
(define (tree-sum t)
  (cond [(leaf? t) (leaf-val t)]
        [else (+ (tree-sum (node-left t))
                 (tree-sum (node-right t)))]))
```

In this module, we have defined two new datatypes: `leaf` and `node`. We've also defined the type name `Tree` to be `(U node leaf)`, which represents a binary tree of numbers. In essence, we are saying that the `tree-height` function accepts a `Tree`, which is either a `node` or a `leaf`, and produces a number.

In order to calculate interesting facts about trees, we have to take them apart and get at their contents. But since accessors such as `node-left` require a `node` as input, not a `Tree`, we have to determine which kind of input we were passed.

For this purpose, we use the predicates that come with each defined structure. For example, the `leaf?` predicate distinguishes `leaf`s from all other Typed Racket values. Therefore, in the first branch of the `cond` clause in `tree-sum`, we know that `t` is a `leaf`, and therefore we can get its value with the `leaf-val` function.

In the else clauses of both functions, we know that `t` is not a `leaf`, and since the type of `t` was `Tree` by process of elimination we can determine that `t` must be a `node`. Therefore, we can use accessors such as `node-left` and `node-right` with `t` as input.

The process by which Typed Racket type-checks the bodies of the `cond` clauses, using information from the predicate checks, is called occurrence typing and is described in detail in §5 "Occurrence Typing".

## 2.2 Type Errors

When Typed Racket detects a type error in the module, it raises an error before running the program.

Example:

```
> (add1 "not a number")
eval:9:0: Type Checker: type mismatch
  expected: Number
  given: String
  in: "not a number"
```

# 3 Specifying Types

The previous section introduced the basics of the Typed Racket type system. In this section, we will see several new features of the language, allowing types to be specified and used.

## 3.1 Type Annotation and Binding Forms

In general, variables in Typed Racket must be annotated with their type. A later subsection (§3.2.1 "When do you need type annotations?") introduces a heuristic which more precisely details when type annotations are needed.

### 3.1.1 Annotating Definitions

We have already seen the `:` type annotation form. This is useful for definitions, at both the top level of a module

```
(: x Number)
(define x 7)
```

and in an internal definition

```
(let ()
  (: x Number)
  (define x 7)
  (add1 x))
```

In addition to the `:` form, almost all binding forms from `racket` are replaced with counterparts which allow the specification of types. Typed Racket's `define` form allows the definition of variables in both top-level and internal contexts.

```
(define x : Number 7)
(define (id [z : Number]) : Number z)
```

Here, `x` has the type `Number`, and `id` has the type `(-> Number Number)`. In the body of `id`, `z` has the type `Number`.

### 3.1.2 Annotating Local Binding

```
(let ([x : Number 7])
  (add1 x))
```

The `let` form is exactly like `let` from `racket`, but type annotations may be provided for each variable bound. Here, `x` is given the type `Number`. The `let*` and `letrec` are similar. Annotations are optional with `let` and variants.

```
(let-values ([([x : Number] [y : String]) (values 7 "hello")])
  (+ x (string-length y)))
```

The `let*-values` and `letrec-values` forms are similar.

### 3.1.3 Annotating Functions

Function expressions also bind variables, which can be annotated with types. This function expects two arguments, a `Number` and a `String`:

```
(lambda ([x : Number] [y : String]) (+ x 5))
```

This function accepts at least one `String`, followed by arbitrarily many `Numbers`. In the body, `y` is a list of `Numbers`.

```
(lambda ([x : String] . [y : Number *]) (apply + y))
```

This function has the type `(-> String Number * Number)`. To specify the return type, add a type annotation after the arguments:

```
(lambda ([x : String] . [y : Number *]) : (U Number String) (apply + y))
```

Functions defined by cases may also be annotated:

```
(case-lambda [() 0]
             [([x : Number]) x])
```

This function has the type `(case-> (-> Number) (-> Number Number))`. To specify the return type, either annotate the entire function or use the expression annotation form (`ann`) inside each case.

### 3.1.4 Annotating Single Variables

When a single variable binding needs annotation, the annotation can be applied to a single variable using a reader extension:

```
(let ([#{x : Number} 7]) (add1 x))
```

This is equivalent to the earlier use of `let`. This is mostly useful for binding forms which do not have counterparts provided by Typed Racket, such as `match`:

```
(: assert-symbols! ((Listof Any) -> (Listof Symbol)))
(define (assert-symbols! lst)
  (match lst
    [(list (? symbol? #{s : (Listof Symbol)}) ...) s]
    [_ (error "expected only symbols, given" lst)]))
```

### 3.1.5 Annotating Expressions

It is also possible to provide an expected type for a particular expression.

```
(ann (+ 7 1) Number)
```

This ensures that the expression, here `(+ 7 1)`, has the desired type, here `Number`. Otherwise, the type checker signals an error. For example:

```
> (ann "not a number" Number)
eval:2:0: Type Checker: type mismatch
  expected: Number
  given: String
  in: Number
```

## 3.2 Type Inference

In many cases, type annotations can be avoided where Typed Racket can infer them. For example, the types of all local bindings using `let` and `let*` can be inferred.

```
(let ([x 7]) (add1 x))
```

In this example, `x` has the type `Exact-Positive-Integer`.

Similarly, top-level constant definitions do not require annotation:

```
(define y "foo")
```

In this examples, `y` has the type `String`.

Finally, the parameter types for loops are inferred from their initial values.

```
(let loop ([x 0] [y (list 1 2 3)])
  (if (null? y) x (loop (+ x (car y)) (cdr y))))
```

Here `x` has the inferred type `Integer`, and `y` has the inferred type `(Listof Integer)`. The `loop` variable has type `(-> Integer (Listof Integer) Integer)`.

### 3.2.1 When do you need type annotations?

The last several subsections explained several ways to add type annotations and explained that type inference allows some annotations to be left out. Since annotations can often be omitted, it is helpful to know the situations in which they are actually required.

The following four rules of thumb will usually suffice to determine if a type annotation is necessary.

An expression or definition needs a type annotation if it:

- is a `define` form for a function,
- is a `lambda` that is immediately bound to a variable,
- is a `lambda` that is an argument to a polymorphic function, or
- is defining a mutable variable.

Here are examples that correspond to each of the cases above:

Example 1:

```
(: fn (-> String Symbol))
(define (fn str) ...)
```

Example 2:

```
(: fn (-> String Symbol))
(define fn (lambda (str) ...))
```

Example 3:

```
(map (lambda ([n : Integer]) (add1 n)) '(1 2 3))
```

Example 4:

```
(: maybe-animal (Option String))
(define maybe-animal #f)
(set! maybe-animal "Odontodactylus scyllarus")
```

In all four cases, if the type annotation is omitted then the inferred type will often be too conservative (e.g., `Any`) and the code may not type-check.

## 3.3 New Type Names

Any type can be given a name with `define-type`.

```
(define-type NN (-> Number Number))
```

Anywhere the name `NN` is used, it is expanded to `(-> Number Number)`. Type names may be recursive or even mutually recursive.

# 4 Types in Typed Racket

Typed Racket provides a rich variety of types to describe data. This section introduces them.

## 4.1 Basic Types

The most basic types in Typed Racket are those for primitive data, such as `True` and `False` for booleans, `String` for strings, and `Char` for characters.

```
> '"hello, world"
- : String
"hello, world"
> #\f
- : Char
#\f
> #t
- : True
#t
> #f
- : False
#f
```

Each symbol is given a unique type containing only that symbol. The `Symbol` type includes all symbols.

```
> 'foo
- : 'foo
'foo
> 'bar
- : 'bar
'bar
```

Typed Racket also provides a rich hierarchy for describing particular kinds of numbers.

```
> 0
- : Integer [more precisely: Zero]
0
> -7
- : Integer [more precisely: Negative-Fixnum]
-7
> 14
- : Integer [more precisely: Positive-Byte]
14
```

```
> 3.2
- : Flonum [more precisely: Positive-Float-No-NaN]
3.2
> 7.0+2.8i
- : Float-Complex
7.0+2.8i
```

Finally, any value is itself a type:

```
> (ann 23 23)
- : Integer [more precisely: 23]
23
```

## 4.2 Function Types

We have already seen some examples of function types. Function types are constructed using `->`, where the last type is the result type and the others are the argument types. Here are some example function types:

```
(-> Number Number)
(-> String String Boolean)
(-> Char (Values String Natural))
```

The first type requires a `Number` as input, and produces a `Number`. The second requires two arguments. The third takes one argument, and produces multiple values, of types `String` and `Natural`. Here are example functions for each of these types.

```
> (lambda ([x : Number]) x)
- : (-> Number Number)
#<procedure>
> (lambda ([a : String] [b : String]) (equal? a b))
- : (-> String String Boolean)
#<procedure>
> (lambda ([c : Char]) (values (string c) (char->integer c)))
- : (-> Char (values (String : (Top | Bot)) (Index : (Top |
Bot))))
#<procedure>
```

## 4.3 Types for Functions with Optional or Keyword Arguments

Racket functions often take optional or keyword arguments in addition to standard mandatory arguments. Types for these functions can be written concisely using the `->*` type constructor. Here are some examples:

```
(->* () (Number) Number)
(->* (String String) Boolean)
(->* (#:x Number) (#:y Number) (values Number Number))
```

The first type describes a function that has no mandatory arguments, one optional argument with type `Number`, and returns a `Number`.

The second requires two mandatory arguments, no optional arguments, and produces a `Boolean`. This function type could have been written using `->` as `(-> String String Boolean)`.

The third requires a mandatory keyword argument with the keyword `#:x` and accepts an optional argument with keyword `#:y`. The result is two values of type `Number`.

## 4.4 Union Types

Sometimes a value can be one of several types. To specify this, we can use a union type, written with the type constructor `U`.

```
> (let ([a-number 37])
    (if (even? a-number)
        'yes
        'no))
- : (U 'no 'yes)
'no
```

Any number of types can be combined together in a union, and nested unions are flattened.

```
(U Number String Boolean Char)
```

## 4.5 Recursive Types

*Recursive types* are types whose definitions refer to themselves. This allows a type to describe an infinite family of data. For example, this is the type of binary trees of numbers.

```
(define-type BinaryTree (U Number (Pair BinaryTree BinaryTree)))
```

Recursive types can also be created anonymously without the use of `define-type` using the `Rec` type constructor.

Types can also be *mutually recursive*. For example, the above type defintion could also be written like this.

```
(define-type BinaryTree (U BinaryTreeLeaf BinaryTreeNode))
(define-type BinaryTreeLeaf Number)
(define-type BinaryTreeNode (Pair BinaryTree BinaryTree))
```

Of course, all recursive types must pass the contractivity check. In other words, types which directly refer to themselves are not permitted. They must be used as arguments to productive type constructors, such as `Listof` and `Pairof`. For example, of the following definitions, only the last is legal.

```
> (define-type BinaryTree BinaryTree)
eval:18:0: Type Checker: Error in macro expansion -- parse
error in type;
 not in a productive position
   variable: BinaryTree
   in: BinaryTree
> (define-type BinaryTree (U Number BinaryTree))
eval:19:0: Type Checker: Error in macro expansion -- parse
error in type;
 not in a productive position
   variable: BinaryTree
   in: BinaryTree
> (define-type BinaryTree (U Number (Listof BinaryTree)))
```

## 4.6 Structure Types

Using `struct` introduces new types, distinct from any previous type.

```
(struct point ([x : Real] [y : Real]))
```

Instances of this structure, such as `(point 7 12)`, have type `point`.

If a struct supertype is provided, then the newly defined type is a subtype of the parent.

## 4.7 Types for Structure Type Properties

To annotate a new structure type property created by `make-struct-type-property`, it must be defined via `define-values` at the top level or module level:

```
> (: prop:foo (Struct-Property (-> Self Number)))
> (: foo-pred (-> Any Boolean : (Has-Struct-Property prop:foo)))
> (: foo-accessor (-> (Has-Struct-Property prop:foo)
                      (Some (X) (-> X Number) : #:+ X)))
> (define-values (prop:foo foo-pred foo-accessor)
                 (make-struct-type-property 'foo))
```

`Struct-Property` creates a type for a structure type property descriptor and its argument is the expected type for property values. In particular, when a structure type property expects

a function to be applied with the receiver, a structure instance the property value is extracted from, `Self` is used to denotes the receiver type. For a value in supplied in a `struct` definition for such a property, we use the structure type for a by-position parameter for `Self`:

```
(struct apple ([a : Number])
   #:property prop:foo
   (lambda ([me : apple]) : Number
       (apple-a me)))
```

A property predicate tells the arguments variable is a `Has-Struct-Property` if the predicate check succeeds. `Has-Struct-Property` describes a subtyping relation between structure types and properties attached to them. In the example above, `apple` is a subtype of `(Has-Struct-Property prop:foo)`

For a property accessor procedure, the argument must have a `Has-Struct-Property` type. If a property expects a value to be a function called with the receiver, i.e. `Self` appears in the type of the corresponding property descriptor, an existential type result is required. Its quantifier needs to correspond to `Self` and also appear in the `proposition`. Such a return type ensures that the extracted function cannot be called with another instance of the structure type or substructure types other than the receiver:

```
> (let ([a1 : apple (apple 42)])
    ((foo-accessor a1) a1))
- : Number
42
> (let ([a1 : apple (apple 42)])
    ((foo-accessor a1) (apple 10)))
eval:27:0: Type Checker: type mismatch
  expected: X
  given: apple
  in: 10
```

Otherwise, the return type should be the same as the type argument to `Struct-Property` for the descriptor.

## 4.8 Subtyping

In Typed Racket, all types are placed in a hierarchy, based on what values are included in the type. When an element of a larger type is expected, an element of a smaller type may be provided. The smaller type is called a *subtype* of the larger type. The larger type is called a *supertype*. For example, `Integer` is a subtype of `Real`, since every integer is a real number. Therefore, the following code is acceptable to the type checker:

```
(: f (-> Real Real))
```

```
(define (f x) (* x 0.75))

(: x Integer)
(define x -125)

(f x)
```

All types are subtypes of the `Any` type.

The elements of a union type are individually subtypes of the whole union, so `String` is a subtype of `(U String Number)`. One function type is a subtype of another if they have the same number of arguments, the subtype's arguments are more permissive (is a supertype), and the subtype's result type is less permissive (is a subtype). For example, `(-> Any String)` is a subtype of `(-> Number (U String #f))`.

## 4.9 Polymorphism

Typed Racket offers abstraction over types as well as values. This allows the definition of functions that use *parametric polymorphism*.

### 4.9.1 Type Constructors

Types for built-in collections are created by built-in type constructors. Users can also define their own type constructors through `define-type`.

Note that types and type constructors are different. If a type constructor is used in a position where a type, the type checker will report a type error:

```
> (ann 10 (Listof Listof))
eval:28:0: Type Checker: Error in macro expansion -- parse
error in type;
 expected a valid type not a type constructor
  given: Listof
  in: Listof
```

Conversely, types cannot be used as type constructors:

```
> (ann 10 (Number Number))
eval:29:0: Type Checker: Error in macro expansion -- parse
error in type;
 bad syntax in type application: expected a type constructor
  given a type: Number
  in: (Number Number)
```

### 4.9.2 Polymorphic Data Structures

Virtually every Racket program uses lists and other collections. Fortunately, Typed Racket can handle these as well. A simple list processing program can be written like this:

```
#lang typed/racket
(: sum-list (-> (Listof Number) Number))
(define (sum-list l)
  (cond [(null? l) 0]
        [else (+ (car l) (sum-list (cdr l)))]))
```

This looks similar to our earlier programs — except for the type of `l`, which looks like a function application. In fact, it's a use of the *type constructor* `Listof`, which takes another type as its input, here `Number`. We can use `Listof` to construct the type of any kind of list we might want.

We can define our own type constructors as well. For example, here is an analog of the `Maybe` type constructor from Haskell:

```
#lang typed/racket
(struct Nothing ())
(struct (A) Just ([v : A]))

(define-type (Maybe A) (U Nothing (Just A)))

(: find (-> Number (Listof Number) (Maybe Number)))
(define (find v l)
  (cond [(null? l) (Nothing)]
        [(= v (car l)) (Just v)]
        [else (find v (cdr l))]))
```

The first `struct` defines `Nothing` to be a structure with no contents.

The second definition

```
(struct (A) Just ([v : A]))
```

creates a type constructor, `Just`, and defines a namesake structure with one element, whose type is that of the type argument to `Just`. Here the type parameters (only one, `A`, in this case) are written before the type name, and can be referred to in the types of the fields.

The type definiton

```
(define-type (Maybe A) (U Nothing (Just A)))
```

creates a type constructor — `Maybe` is a potential container for whatever type is supplied.

The `find` function takes a number `v` and list, and produces `(Just v)` when the number is found in the list, and `(Nothing)` otherwise. Therefore, it produces a `(Maybe Number)`, just as the annotation specified.

### 4.9.3 Polymorphic Functions

Sometimes functions over polymorphic data structures only concern themselves with the form of the structure. For example, one might write a function that takes the length of a list of numbers:

```
#lang typed/racket
(: list-number-length (-> (Listof Number) Integer))
(define (list-number-length l)
  (if (null? l)
      0
      (add1 (list-number-length (cdr l)))))
```

and also a function that takes the length of a list of strings:

```
#lang typed/racket
(: list-string-length (-> (Listof String) Integer))
(define (list-string-length l)
  (if (null? l)
      0
      (add1 (list-string-length (cdr l)))))
```

Notice that both of these functions have almost exactly the same definition; the only difference is the name of the function. This is because neither function uses the type of the elements in the definition.

We can abstract over the type of the element as follows:

```
#lang typed/racket
(: list-length (All (A) (-> (Listof A) Integer)))
(define (list-length l)
  (if (null? l)
      0
      (add1 (list-length (cdr l)))))
```

The new type constructor `All` takes a list of type variables and a body type. The type variables are allowed to appear free in the body of the `All` form.

### 4.9.4 Lexically Scoped Type Variables

When the `:` type annotation form includes type variables for parametric polymorphism, the type variables are *lexically scoped*. In other words, the type variables are bound in the body of the definition that you annotate.

For example, the following definition of *my-id* uses the type variable `a` to annotate the argument *x*:

```
(: my-id (All (a) (-> a a)))
(define my-id (lambda ([x : a]) x))
```

Lexical scope also implies that type variables can be shadowed, such as in the following example:

```
(: my-id (All (a) (-> a a)))
(define my-id
  (lambda ([x : a])
    (: helper (All (a) (-> a a)))
    (define helper
      (lambda ([y : a]) y))
    (helper x)))
```

The reference to `a` inside the inner `lambda` refers to the type variable in *helper*'s annotation. That `a` is *not* the same as the `a` in the annotation of the outer `lambda` expression.

## 4.10 Variable-Arity Functions: Programming with Rest Arguments

Typed Racket can handle some uses of rest arguments.

### 4.10.1 Uniform Variable-Arity Functions

In Racket, one can write a function that takes an arbitrary number of arguments as follows:

```
#lang racket
(define (sum . xs)
  (if (null? xs)
      0
      (+ (car xs) (apply sum (cdr xs)))))

(sum)
(sum 1 2 3 4)
(sum 1 3)
```

The arguments to the function that are in excess to the non-rest arguments are converted to a list which is assigned to the rest parameter. So the examples above evaluate to 0, 10, and 4.

We can define such functions in Typed Racket as well:

```
#lang typed/racket
(: sum (-> Number * Number))
(define (sum . xs)
  (if (null? xs)
      0
      (+ (car xs) (apply sum (cdr xs)))))
```

This type can be assigned to the function when each element of the rest parameter is used at the same type.

### 4.10.2 Non-Uniform Variable-Arity Functions

However, the rest argument may be used as a heterogeneous list. Take this (simplified) definition of the R6RS function `fold-left`:

```
#lang racket
(define (fold-left f i as . bss)
  (if (or (null? as)
          (ormap null? bss))
      i
      (apply fold-left
             f
             (apply f i (car as) (map car bss))
             (cdr as)
             (map cdr bss))))

(fold-left + 0 (list 1 2 3 4) (list 5 6 7 8))
(fold-left + 0 (list 1 2 3) (list 2 3 4) (list 3 4 5) (list 4 5 6))
(fold-left (λ (i v n s) (string-append i (vector-ref v n) s))
           ""
           (list (vector "A cat" "A dog" "A mouse")
                 (vector "tuna" "steak" "cheese"))
           (list 0 2)
           (list " does not eat " "."))
```

Here the different lists that make up the rest argument `bss` can be of different types, but the type of each list in `bss` corresponds to the type of the corresponding argument of `f`. We also know that, in order to avoid arity errors, the length of `bss` must be two less than the arity of

`f`. The first argument to `f` is the accumulator, and `as` corresponds to the second argument of `f`.

The example uses of `fold-left` evaluate to `36`, `42`, and `"A cat does not eat cheese."`.

In Typed Racket, we can define `fold-left` as follows:

```
#lang typed/racket
(: fold-left
   (All (C A B ...)
        (-> (-> C A B ... B C) C (Listof A) (Listof B) ... B
            C)))
(define (fold-left f i as . bss)
  (if (or (null? as)
          (ormap null? bss))
      i
      (apply fold-left
             f
             (apply f i (car as) (map car bss))
             (cdr as)
             (map cdr bss))))
```

Note that the type variable `B` is followed by an ellipsis. This denotes that B is a dotted type variable which corresponds to a list of types, much as a rest argument corresponds to a list of values. When the type of `fold-left` is instantiated at a list of types, then each type `t` which is bound by `B` (notated by the dotted pre-type `t ... B`) is expanded to a number of copies of `t` equal to the length of the sequence assigned to `B`. Then `B` in each copy is replaced with the corresponding type from the sequence.

So the type of `(inst fold-left Integer Boolean String Number)` is

`(-> (-> Integer Boolean String Number Integer) Integer (Listof Boolean) (Listof String) (Listof Number) Integer)`.

# 5 Occurrence Typing

## 5.1 Basic Occurrence Typing

One of Typed Racket's distinguishing type system features is *occurrence typing*, which allows the type system to ascribe more precise types based on whether a predicate check succeeds or fails.

To illustrate, consider the following code:

```
(: flexible-length (-> (U String (Listof Any)) Integer))
(define (flexible-length str-or-lst)
  (if (string? str-or-lst)
      (string-length str-or-lst)
      (length str-or-lst)))
```

The *flexible-length* function above computes the length of either a string or a list. The function body uses the typical Racket idiom of dispatching using a predicate (e.g., `string?`).

Typed Racket successfully type-checks this function because the type system understands that in the "then" branch of the `if` expression, the predicate `string?` must have returned a true value. The type system further knows that if `string?` returns true, then the *str-or-lst* variable must have type `String` and can narrow the type from its original union of `String` and `(Listof Any)`. This allows the call to `string-length` in the "then" branch to type-check successfully.

Furthermore, the type system also knows that in the "else" branch of the `if` expression, the predicate must have returned `#f`. This implies that the variable *str-or-lst* must have type `(Listof Any)` by process of elimination, and thus the call `(length str-or-lst)` type-checks.

To summarize, if Typed Racket can determine the type a variable must have based on a predicate check in a conditional expression, it can narrow the type of the variable within the appropriate branch of the conditional.

## 5.2 Propositions and Predicates

In the previous section, we demonstrated that a Typed Racket programmer can take advantage of occurrence typing to type-check functions with union types and conditionals. This may raise the question: how does Typed Racket know how to narrow the type based on the predicate?

The answer is that predicate types in Typed Racket are annotated with logical *propositions*

that tell the typechecker what additional information is gained when a predicate check succeeds or fails.

For example, consider the REPL's type printout for `string?`:

```
> string?
- : (-> Any Boolean : String)
#<procedure:string?>
```

The type `(-> Any Boolean : String)` has three parts. The first two are the same as any other function type and indicate that the predicate takes any value and returns a boolean. The third part, after the `:`, represents the logical propositions the typechecker learns from the result of applying the function:

1. If the predicate check succeeds (i.e. produces a non-`#f` value), the argument variable has type `String`
2. If the predicate check fails (i.e. produces `#f`), the argument variable *does not* have type `String`

Predicates for all built-in types are annotated with similar propositions that allow the type system to reason logically about predicate checks.

### 5.2.1 Specifying Propositions

While propositions are provided for all built-in type predicates, we may want to provide propositions for our own predicates as well. For instance, consider the following predicate, which determines whether a given list contains only strings. Intuitively, a value that satisfies the predicate must have type `(Listof String)`.

```
(: listof-string? (-> (Listof Any) Boolean))
(define (listof-string? lst)
  (andmap string? lst))
```

We then may wish to use this predicate to narrow a type in the `main` function:

```
> (: main (-> (Listof Any) String))
> (define (main lst)
    (cond
      [(listof-string? lst) (first lst)]
      [else "not a list of strings"]))
eval:5:0: Type Checker: Polymorphic function `first' could
not be applied to arguments:
```

```
Types: (Pairof a (Listof b))    -> (a : ((! (car (0 0)) False)
| (: (car (0 0)) False)) : (car (0 0)))
          (Listof a)    -> a
Arguments: (Listof Any)
Expected result: String

   in: "not a list of strings"
```

Unfortunately, Typed Racket fails to narrow the type, because we did not specify a proposition for `listof-string?`. To fix this issue, we include the proposition in the `->` form for `listof-string?`.

```
(: listof-string? (-> (Listof Any) Boolean : (Listof String)))
(define (listof-string? lst)
  (andmap string? lst))
```

Note that if we directly use `(andmap string? lst)` as the conditional expression, `main` would be successfully type-checked, because `andmap` and `string?` do provide propositions that allow Typed Racket to narrow the type.

With the proposition, Typed Racket successfully type-checks `main`.

```
> (: main (-> (Listof Any) String))
> (define (main lst)
    (cond
      [(listof-string? lst) (first lst)]
      [else "not a list of strings"]))
```

### 5.2.2 One-sided Propositions

Sometimes, a predicate may provide information when it succeeds, but not when it fails. For instance, consider this function:

```
(define (legal-id? s)
  (and (symbol? s)
       (not (member s '(cond else if)))))
```

This function only returns `#t` when given a symbol, so the type of something that satisfies this predicate can be refined to Symbol.

However, values that fail this predicate can't be refined to non-symbols; symbols such as `'else` also fail to satisfy this predicate.

In cases such as these, it's possible to provide a proposition that's applied only to the "positive" assertion. Specifically, this type

```
(: legal-id? (Any -> Boolean : #:+ Symbol))
```

... captures the idea that if this predicate returns `#t`, the argument is known to be a Symbol, without making any claim at all about values for which this predicate returns `#f`.

There is a negative form as well, which allows types that specify propositions only about values that cause a predicate to return `#f`.

## 5.3 Other conditionals and assertions

After all, these control flow constructs macro-expand to `if` in the end.

So far, we have seen that occurrence typing allows precise reasoning about `if` expressions. Occurrence typing works for most control flow constructs that are present in Racket such as `cond`, `when`, and others.

For example, the *`flexible-length`* function from earlier can be re-written to use `cond` with no additional effort:

```
(: flexible-length/cond (-> (U String (Listof Any)) Integer))
(define (flexible-length/cond str-or-lst)
  (cond [(string? str-or-lst) (string-length str-or-lst)]
        [else (length str-or-lst)]))
```

In some cases, the type system does not have enough information or is too conservative to type-check an expression. For example, consider the following interaction:

```
> (: a Positive-Integer)
> (define a 15)
> (: b Positive-Integer)
> (define b 20)
> (: c Positive-Integer)
> (define c (- b a))
eval:17:0: Type Checker: type mismatch
  expected: Positive-Integer
  given: Integer
  in: a
```

In this case, the type system only knows that `a` and `b` are positive integers and cannot conclude that their difference will always be positive in defining `c`. In cases like this, occurrence typing can be used to make the code type-check using an *assertion*. For example,

```
(: d Positive-Integer)
(define d (assert (- b a) positive?))
```

Using the logical propositions on `positive?`, Typed Racket can assign the type `Positive-Integer` to the whole `assert` expression. This type-checks, but note that the assertion may raise an exception at run-time if the predicate returns `#f`.

Note that `assert` is a derived concept in Typed Racket and is a natural consequence of occurrence typing. The assertion above is essentially equivalent to the following:

```
(: e Positive-Integer)
(define e (let ([diff (- b a)])
            (if (positive? diff)
                diff
                (error "Assertion failed"))))
```

## 5.4 A caveat about `set!`

If a variable is ever mutated with `set!` in the scope in which it is defined, Typed Racket cannot use occurrence typing with that variable. This precaution is needed to ensure that concurrent modification of a variable does not invalidate Typed Racket's knowledge of the type of that variable. Also see §4.9.1 "Guidelines for Using Assignment".

Furthermore, this means that the types of top-level variables in the REPL cannot be refined by Typed Racket either. This is because the scope of a top-level variable includes future top-level interactions, which may include mutations. It is possible to work around this by moving the variable inside of a module or into a local binding form like `let`.

## 5.5 Access to structure fields

Occurrence typing can work with accessors to immutable structure fields.

```
(struct apple ([a : Any]))
(struct (A) fruit ([a : A]))

(define (f [obj : Any]) : Number
  (cond
    [(and (apple? obj) (number? (apple-a obj))) (apple-a obj)]
    [(and (fruit? obj) (number? (fruit-a obj))) (fruit-a obj)]
    [else 42]))
```

## 5.6 `let`-aliasing

Typed Racket is able to reason about some cases when variables introduced by let-expressions alias other values (e.g. when they alias non-mutated identifiers, `car`/`cdr`/`struct` accesses into immutable values, etc...). This allows programs which explicitly rely on occurrence typing and aliasing to type-check:

```
(: f (Any -> Number))
```

```
(define (f x)
  (let ([y x])
    (cond
      [(number? y) x]
      [(and (pair? y)
            (number? (car y)))
       (car x)]
      [else 42])))
```

It also allows the typechecker to check programs which use macros that heavily rely on let-bindings internally (such as `match`):

```
(: g (Any -> Number))
(define (g x)
  (match x
    [(? number?) x]
    [`(_    _ . ,(? number?))  (cddr x)]
    [`(_    _ . ,(? pair? p))
     (if (number? (caddr x))
         (car p)
         41)]
    [_ 42]))
```

# 6 Typed-Untyped Interaction

In the previous sections, all of the examples have consisted of programs that are entirely typed. One of the key features of Typed Racket is that it allows the combination of both typed and untyped code in a single program.

From a static typing perspective, combining typed and untyped code is straightforward. Typed code must declare types for its untyped imports to let the type checker validate their use (§6.1 "Using Untyped Code in Typed Code"). Untyped code can freely import bindings from typed code (§6.2 "Using Typed Code in Untyped Code").

At run-time, combining typed and untyped code is complicated because there is a tradeoff between strong type guarantees and the performance cost of checking that untyped code matches the types. Typed Racket provides strong *Deep* type guarantees by default, but offers two weaker options as well: *Shallow* and *Optional* types (§6.3 "Protecting Typed-Untyped Interaction").

## 6.1 Using Untyped Code in Typed Code

Suppose that we write the untyped module from §1 "Quick Start" again:

`"distance.rkt"`

```
#lang racket

(provide (struct-out pt)
         distance)

(struct pt (x y))

; distance : pt pt -> real
(define (distance p1 p2)
  (sqrt (+ (sqr (- (pt-x p2) (pt-x p1)))
           (sqr (- (pt-y p2) (pt-y p1))))))
```

If we want to use the *distance* function defined in the above module from a typed module, we need to use the `require/typed` form to import it. Since the untyped module did not specify any types, we need to annotate the imports with types (just like how the example in §1 "Quick Start" had additional type annotations with `:`):

`"client.rkt"`

```
#lang typed/racket

(require/typed "distance.rkt"
               [#:struct pt ([x : Real] [y : Real])]
```

Note that a typed module *should not* use `require/typed` to import from another typed module. The `require` form will work in such cases.

```
                [distance (-> pt pt Real)])

(distance (pt 3 5) (pt 7 0))
```

The `require/typed` form has several kinds of clauses. The `#:struct` clause specifies the import of a structure type and allows us to use the structure type as if it were defined with Typed Racket's `struct`.

The second clause in the example above specifies that a given binding *distance* has the given type `(-> pt pt Real)`.

Note that the `require/typed` form can import bindings from any module, including those that are part of the Racket standard library. For example,

```
#lang typed/racket

(require/typed racket/base [add1 (-> Integer Integer)])
```

is a valid use of the `require/typed` form and imports `add1` from the `racket/base` library.

### 6.1.1 Opaque Types

The `#:opaque` clause of `require/typed` defines a new type using a predicate from untyped code. Suppose we have an untyped distance function that uses pairs of numbers as points:

"distance2.rkt"

```
#lang racket

(provide point?
         distance)

; A Point is a (cons real real)
(define (point? x)
  (and (pair? x)
       (real? (car x))
       (real? (cdr x))))

; distance : Point Point -> real
(define (distance p1 p2)
  (sqrt (+ (sqr (- (car p2) (car p1)))
           (sqr (- (cdr p2) (cdr p1))))))
```

A typed module can use `#:opaque` to define a `Point` type as all values that the `point?` predicate returns `#t` for:

"client2.rkt"

```
#lang typed/racket

(require/typed "distance2.rkt"
               [#:opaque Point point?]
               [distance (-> Point Point Real)])

(define p0 : Point (assert (cons 3 5) point?))
(define p1 : Point (assert (cons 7 0) point?))
(distance p0 p1)
```

## 6.2 Using Typed Code in Untyped Code

In the previous subsection, we saw that the use of untyped code from typed code requires the use of `require/typed`. However, the use of code in the other direction (i.e., the use of typed code from untyped code) requires no additional work.

If an untyped module `requires` a typed module, it will be able to use the bindings defined in the typed module as expected. The major exception to this rule is that macros defined in typed modules may not be used in untyped modules.

## 6.3 Protecting Typed-Untyped Interaction

One might wonder if the interactions described in the first two subsections are actually safe. After all, untyped code might be able to ignore the errors that Typed Racket's type system will catch at compile-time.

For example, suppose that we write an untyped module that implements an *increment* function:

Example:

```
> (module increment racket
    (provide increment)

    ; increment : exact-integer? -> exact-integer?
    (define (increment x) "this is broken"))
```

and a typed module that uses it:

```
> (module client typed/racket

    (require/typed 'increment [increment (-> Integer Integer)])
```

```
(increment 5))
```

This combined program has a problem. All uses of *increment* in Typed Racket are correct under the assumption that the *increment* function upholds the `(-> Integer Integer)` type. Unfortunately, our *increment* implementation does not actually uphold this assumption, because the function actually produces strings.

By default, Typed Racket establishes contracts wherever typed and untyped code interact to ensure strong types. These contracts can, however, have a non-trivial performance impact. For programs in which these costs are problematic, Typed Racket provides two alternatives. All together, the three options are Deep, Shallow, and Optional types.

1. Deep types get enforced with rigorous contract checks.
2. Shallow types get checked in typed code with lightweight assertions called shape checks.
3. Optional types do not get enforced in any way. They do not ensure safe typed-untyped interactions.

See also: §8 "Deep, Shallow, and Optional Semantics" in the Typed Racket Reference.

The next subsections give examples of Deep, Shallow, and Optional behaviors.

### 6.3.1 Deep Types: Completely Reliable

When the *client* program above is run, standard Typed Racket (aka. Deep Typed Racket) enforces the `require/typed` interface with a contract. This contract detects a failed type assumption when the *client* calls the untyped *increment* function:

```
> (require 'client)
increment: broke its own contract
  promised: exact-integer?
  produced: "this is broken"
  in: (-> any/c exact-integer?)
  contract from: (interface for increment)
  blaming: (interface for increment)
   (assuming the contract is correct)
  at: eval:3:0
```

Because the implementation in the untyped module broke the contract by returning a string instead of an integer, the error message blames it.

For general information on Racket's contract system, see §7 "Contracts".

In general, Deep Typed Racket checks all functions and other values that pass from a typed module to untyped module or vice versa with contracts. This means that, for example,

Typed Racket can safely optimize programs (see §7 “Optimization in Typed Racket”) with the assurance that the program will not segfault due to an unchecked assumption.

**Important caveat**: contracts such as the `Integer` check from above are performant. However, contracts in general can have a non-trivial performance impact, especially with the use of first-class functions or other higher-order data such as vectors.

Note that no contract overhead is ever incurred for uses of typed values from another Deep-typed module.

### 6.3.2 Shallow Types: Sound Types, Low-Cost Interactions

Changing the module language of the *client* program from `typed/racket` to `typed/racket/shallow` changes the way in which typed-untyped interactions are protected. Instead of contracts, Typed Racket uses *shape checks* to enforce these Shallow types.

With Shallow types, the *client* program from above still detects an error when an untyped function returns a string instead of an integer:

```
> (module client typed/racket/shallow

    (require/typed 'increment [increment (-> Integer Integer)])

    (increment 5))
> (require 'client)
shape-check: value does not match expected type
  value: "this is broken"
  type: Integer
  lang: 'typed/racket/shallow
  src: '(eval 2 0 2 1)
```

The compiled *client* module has two shape checks in total:

1. A shape check at the `require/typed` boundary confirms that `increment` is a function that expects one argument.
2. A shape check after the call `(increment 5)` looks for an integer. This check fails.

Such checks work together within one typed module to enforce the assumptions that it makes about untyped code.

A design guideline for a shape checks is to ensure that a value matches the top-level constructor of a type. Shape checks are always yes-or-no predicates (unlike contracts, which may wrap a value) and typically run in constant time. Because they ensure the validity of type

constructors, shape checks allow Typed Racket to safely optimize some programs—though not to the same extent as Deep types.

**Important caveats**: (1) The number of shape checks in a module grows in proportion to its size. For example, every function call in Shallow-typed code gets checked—unless Typed Racket is certain that it can trust the function. Shallow types are therefore a poor choice for large, computationally-heavy modules. (2) Shallow types are only enforced in their immediate, local context. For example, if typed code were to cast `increment` to expect a string, then the function could be called without an error.

### 6.3.3 Optional Types: It's Just Racket

A third option for the *client* program is to use Optional types, which are provided by the language `typed/racket/optional`:

```
> (module client typed/racket/optional

    (require/typed 'increment [increment (-> Integer Integer)])

    (increment 5))
```

Optional types do not ensure safe typed-untyped interactions. In fact, they do nothing to check types at run-time. A call to the increment function does not raise an error:

```
> (require 'client)
```

Optional types cannot detect incorrect type assumptions and therefore do not enable type-driven optimizations. But, they also add no costs to slow a program down. The run-time behavior is very similar to untyped Racket and `typed/racket/no-check`.

### 6.3.4 When to Use Deep, Shallow, or Optional?

- Deep types maximize the benefits of static checking and type-driven optimizations. Use them for tightly-connected groups of typed modules. Avoid them when untyped, higher-order values frequently cross boundaries into typed code. Expensive boundary types include `Vectorof`, `->`, and `Object`.
- Shallow types are best for small typed modules that frequently interact with untyped code. This is because Shallow shape checks run quickly: constant-time for most types, and linear time (in the size of the type, not the value) for a few exceptions such as `U` and `case->`. Avoid Shallow types in large typed modules that frequently call functions or access data structures because these operations may incur shape checks and their net cost may be significant.

- Optional types enable the typechecker and nothing else. Use them when you do not want types enforced at run-time.

# 7 Optimization in Typed Racket

For general information on Racket performance and benchmarking, see §19 “Performance”.

Typed Racket provides a type-driven optimizer that rewrites well-typed programs to potentially make them faster.

## 7.1 Turning the optimizer off

Typed Racket’s optimizer is turned on by default. If you want to deactivate it (for debugging, for instance), you must add the `#:no-optimize` keyword when specifying the language of your program:

```
#lang typed/racket #:no-optimize
```

The optimizer is also disabled when executing a typed racket program in a sandbox (see §14.12 “Sandboxed Evaluation”) and when the environment variable `PLT_TR_NO_OPTIMIZE` is set (to any value).

## 7.2 Getting the most out of the optimizer

Typed Racket’s optimizer can improve the performance of various common Racket idioms. However, it does a better job on some idioms than on others. By writing your programs using the right idioms, you can help the optimizer help you.

To best take advantage of the Typed Racket optimizer, consult the Optimization Coach documentation.

The Typed Racket optimizer logs events with the topic `'TR-optimizer`. See §15.5 “Logging” to learn how to receive these log events.

### 7.2.1 Numeric types

Being type-driven, the optimizer makes most of its decisions based on the types you assigned to your data. As such, you can improve the optimizer’s usefulness by writing informative types.

For example, the following programs both typecheck:

```
(define (f [x : Real]) : Real (+ x 2.5))
(f 3.5)
```

```
(define (f [x : Float]) : Float (+ x 2.5))
(f 3.5)
```

However, the second one uses more informative types: the `Float` type includes only 64-bit floating-point numbers whereas the `Real` type includes both exact and inexact real numbers and the `Inexact-Real` type includes both 32- and 64-bit floating-point numbers. Typed Racket's optimizer can optimize the latter program to use float -specific operations whereas it cannot do anything with the former program.

Thus, to get the most of Typed Racket's optimizer, you should use the `Float` type when possible. For similar reasons, you should use floating-point literals instead of exact literals when doing floating-point computations.

When mixing floating-point numbers and exact reals in arithmetic operations, the result is not necessarily a `Float`. For instance, the result of `(* 2.0 0)` is `0` which is not a `Float`. This can result in missed optimizations. To prevent this, when mixing floating-point numbers and exact reals, coerce exact reals to floating-point numbers using `real->double-flonum`. This is not necessary when using `+` or `-`. When mixing floating-point numbers of different precisions, results use the highest precision possible.

On a similar note, the `Float-Complex` type is preferable to the `Complex` type for the same reason. Typed Racket can keep float complex numbers unboxed; as such, programs using complex numbers can have better performance than equivalent programs that represent complex numbers as two real numbers. As with floating-point literals, float complex literals (such as `1.0+1.0i`) should be preferred over exact complex literals (such as `1+1i`).

To get the most of Typed Racket's optimizer, you should also favor rectangular coordinates over polar coordinates.

Note that on Racket BC, it is possible to have complex numbers where one component is exact and the other is inexact. These values, including literals written `+1.0i`, do not have the type `Float-Complex`. On Racket CS, such mixed-exactness values do not exist.

### 7.2.2 Lists

Typed Racket handles potentially empty lists and lists that are known to be non-empty differently: when taking the `car` or the `cdr` of a list Typed Racket knows is non-empty, it can skip the check for the empty list that is usually done when calling `car` and `cdr`.

```
(define (sum [l : (Listof Integer)]) : Integer
  (if (null? l)
      0
      (+ (car l) (sum (cdr l)))))
```

In this example, Typed Racket knows that if we reach the else branch, `l` is not empty. The

checks associated with `car` and `cdr` would be redundant and are eliminated.

In addition to explicitly checking for the empty list using `null?`, you can inform Typed Racket that a list is non-empty by using the known-length list type constructor; if your data is stored in lists of fixed length, you can use the `List` type constructors.

For instance, the type of a list of two `Integers` can be written either as:

```
(define-type List-2-Ints (Listof Integer))
```

or as the more precise:

```
(define-type List-2-Ints (List Integer Integer))
```

Using the second definition, all `car` and `cdr`-related checks can be eliminated in this function:

```
(define (sum2 [l : List-2-Ints]) : Integer
  (+ (car l) (car (cdr l))))
```

### 7.2.3 Vectors

In addition to known-length lists, Typed Racket supports known-length vectors through the `Vector` type constructor. Known-length vector access using constant indices can be optimized in a similar fashion as `car` and `cdr`.

```
; #(color r g b)
(define-type Color (Vector String Integer Integer Integer))
(define x : Color (vector "red" 255 0 0))
(vector-ref x 0) ; good
(define color-name 0)
(vector-ref x color-name) ; good
(vector-ref x (* 0 10)) ; bad
```

In many such cases, however, structs are preferable to vectors. Typed Racket can optimize struct access in all cases.

### 7.2.4 Contract boundaries

When interoperating with untyped code (see §6 “Typed-Untyped Interaction”), contracts are installed between typed and untyped modules. Contracts can have significant overhead, thus typed-untyped boundary crossings should be avoided in performance-sensitive code.

Typed Racket provides types for most of the bindings provided by `#lang racket`; using `require/typed` is unnecessary in these cases.

If you suspect that contracts at a typed-untyped boundary may have a significant cost in your program, you can investigate further using the contract profiler.

If the contract profiler is not already installed, the following command will install it:

```
raco pkg install contract-profile
```

# 8 Caveats and Limitations

This section describes limitations and subtle aspects of the type system that programmers often stumble on while porting programs to Typed Racket.

## 8.1 The `Integer` type and `integer?`

In Typed Racket, the `Integer` type corresponds to values that return `#t` for the `exact-integer?` predicate, ***not*** the `integer?` predicate. In particular, values that return `#t` for `integer?` may be inexact numbers (e.g, `1.0`).

When porting a program to Typed Racket, you may need to replace uses of functions like `round` and `floor` with corresponding exact functions like `exact-round` and `exact-floor`.

In other cases, it may be necessary to use `assert`ions or `casts`.

## 8.2 Type inference for polymorphic functions

Typed Racket's local type inference algorithm is currently not able to infer types for polymorphic functions that are used on higher-order arguments that are themselves polymorphic.

For example, the following program results in a type error that demonstrates this limitation:

```
> (map cons '(a b c d) '(1 2 3 4))
eval:2:0: Type Checker: Polymorphic function `map' could not
be applied to arguments:
Domains: (-> a b ... b c) (Listof a) (Listof b) ... b
         (-> a c) (Pairof a (Listof a))
Arguments: (All (a b) (case-> (-> a (Listof a) (Listof a))
(-> a b (Pairof a b)))) (List 'a 'b 'c 'd) (List One
Positive-Byte Positive-Byte Positive-Byte)

  in: 4
```

The issue is that the type of `cons` is also polymorphic:

```
> cons
- : (All (a b) (case-> (-> a (Listof a) (Listof a)) (-> a b
(Pairof a b))))
#<procedure:cons>
```

To make this expression type-check, the `inst` form can be used to instantiate the polymorphic argument (e.g., `cons`) at a specific type:

```
> (map (inst cons Symbol Integer) '(a b c d) '(1 2 3 4))
- : (Listof (Pairof Symbol Integer))
'((a . 1) (b . 2) (c . 3) (d . 4))
```

## 8.3 Typed-untyped interaction and contract generation

When a typed module `requires` bindings from an untyped module (or vice-versa), there are some types that cannot be converted to a corresponding contract.

This could happen because a type is not yet supported in the contract system, because Typed Racket's contract generator has not been updated, or because the contract is too difficult to generate. In some of these cases, the limitation will be fixed in a future release.

The following illustrates an example type that cannot be converted to a contract:

```
> (require/typed racket/base
    [object-name (case-> (-> Struct-Type-Property Symbol)
                         (-> Regexp (U String Bytes)))])
eval:5:0: Type Checker: Error in macro expansion -- Type
(case-> (-> Struct-Type-Property Symbol) (-> Regexp (U Bytes
String))) could not be converted to a contract: function
type has two cases of arity 1
  in: (case-> (-> Struct-Type-Property Symbol) (-> Regexp (U
String Bytes)))
```

This function type by cases is a valid type, but a corresponding contract is difficult to generate because the check on the result depends on the check on the domain. In the future, this may be supported with dependent contracts.

A more approximate type will work for this case, but with a loss of type precision at use sites:

```
> (require/typed racket/base
    [object-name (-> (U Struct-Type-Property Regexp)
                     (U String Bytes Symbol))])
> (object-name #rx"a regexp")
- : (U Bytes String Symbol)
"a regexp"
```

Use of `define-predicate` also involves contract generation, and so some types cannot have predicates generated for them. The following illustrates a type for which a predicate can't be generated:

```
> (define-predicate p? (All (A) (Listof A)))
```

*eval:8:0: Type Checker: Error in macro expansion -- Type (All (A) (Listof A)) could not be converted to a predicate: cannot generate contract for non-function polymorphic type*
*in: (All (A) (Listof A))*

## 8.4 Unsupported features

Typed Racket currently does not support generic interfaces.

## 8.5 Type generalization

Not so much a caveat as a feature that may have unexpected consequences. To make programming with invariant type constructors (such as `Boxof`) easier, Typed Racket generalizes types that are used as arguments to invariant type constructors. For example:

```
> 0
- : Integer [more precisely: Zero]
0
> (define b (box 0))
> b
- : (Boxof Integer)
'#&0
```

`0` has type `Zero`, which means that `b` "should" have type `(Boxof Zero)`. On the other hand, that type is not especially useful, as it only allows `0` to be stored in the box. Most likely, the intent was to have a box of a more general type (such as `Integer`) and initialize it with `0`. Type generalization does exactly that.

In some cases, however, type generalization can lead to unexpected results:

```
> (box (ann 1 Fixnum))
- : (Boxof Integer)
'#&1
```

The intent of this code may be to create of box of `Fixnum`, but Typed Racket will generalize it anyway. To create a box of `Fixnum`, the box itself should have a type annotation:

```
> (ann (box 1) (Boxof Fixnum))
- : (Boxof Fixnum)
'#&1
> ((inst box Fixnum) 1)
- : (Boxof Fixnum)
'#&1
```

## 8.6 Macros and compile-time computation

Typed Racket will type-check all expressions at the run-time phase of the given module and will prevent errors that would occur at run-time. However, expressions at compile-time—including computations that occur inside macros—are not checked.

Concretely, this means that expressions inside, for example, a `begin-for-syntax` block are not checked:

```
> (begin-for-syntax (+ 1 "foo"))
+: contract violation
  expected: number?
  given: "foo"
```

Similarly, expressions inside of macros defined in Typed Racket are not type-checked. On the other hand, the macro's expansion is always type-checked:

```
(define-syntax (example-1 stx)
  (+ 1 "foo")
  #'1)

(define-syntax (example-2 stx)
  #'(+ 1 "foo"))

> (example-1)
+: contract violation
  expected: number?
  given: "foo"
> (example-2)
eval:17:0: Type Checker: type mismatch
  expected: Number
  given: String
  in: (quote "foo")
```

Note that functions defined in Typed Racket that are used at compile-time in other typed modules or untyped modules will be type-checked and then protected with contracts as described in §6 "Typed-Untyped Interaction".

Additionally, macros that are defined in Typed Racket modules cannot be used in ordinary Racket modules because such uses can circumvent the protections of the type system.

## 8.7 Expensive contract boundaries

Contract boundaries installed for typed-untyped interaction may cause significant slowdowns. See §7.2.4 "Contract boundaries" for details.

## 8.8 Pattern Matching and Occurrence Typing

Because Typed Racket type checks code *after* macro expansion, certain forms—such as `match`—are difficult for Typed Racket to reason about completely. In particular, in a `match` clause, the type of an identifier is often *not* updated to reflect the fact that a previous pattern failed to match. For example, in the following function, the type checker is unaware that if execution reaches the last clause then the `string?` predicate has already failed to match on the value for `x`, and so `(abs x)` in the last clause fails to type check:

```
> (: size (-> (U String Integer) Integer))
> (define (size x)
    (match x
      [(? string?) (string-length x)]
      [_ (abs x)]))
eval:21:0: Type Checker: type mismatch
  expected: Integer
  given: (U Integer String)
  in: x
```

Because they are much simpler forms, similar `cond` and `if` expressions do type check successfully:

```
> (: size (-> (U String Integer) Integer))
> (define (size x)
      (cond
        [(string? x) (string-length x)]
        [else (abs x)]))
```

One work around is to simply not rely on a catch-all "else" clause that needs to know that previous patterns have failed to match in order to type check:

```
> (: size (-> (U String Integer) Integer))
> (define (size x)
      (match x
        [(? string?) (string-length x)]
        [(? exact-integer?) (abs x)]))
```

It is important to note, however, that `match` *always* inserts a catch-all failure clause if one is not provided! This means that the type checker will not inform the programmer that match clause coverage is insufficient because the implicit (i.e. macro-inserted) failure clause *will* cover any cases the programmer failed to anticipate with their pattern matching, e.g.:

```
> (: size (-> (U String Integer) Integer))
> (define (size x)
      (match x
        [(? string?) (string-length x)]))
```

```
> (size 42)
match: no matching clause for 42
```

Patterns involving an ellipsis `...` for repetition may generate a `for` loop that requires annotations on variables to type check. The (deliberately obscure) code below does not type check without the type annotation on the match pattern variable `c`.

```
(: do-nothing (-> (Listof Integer) (Listof Integer)))
(define (do-nothing lst)
  (match lst
    [(list (? number? #{c : (Listof Integer)}) ...)   c]))
```

## 8.9 `is-a?` and Occurrence Typing

Typed Racket does not use the `is-a?` predicate to refine object types because the target object may have been created in untyped code and `is-a?` does not check the types of fields and methods.

For example, the code below defines a class type `Pizza%`, a subclass type `Sauce-Pizza%`, and a function `get-sauce` (this function contains a type error). The `get-sauce` function uses `is-a?` to test the class of its argument; if the test is successful, the function expects the argument to have a field named `topping` that contains a value of type `Sauce`.

```
#lang typed/racket

(define-type Pizza%
  (Class (field [topping Any])))

(define-type Sauce
  (U 'tomato 'bbq 'no-sauce))

(define-type Sauce-Pizza%
  (Class #:implements Pizza% (field [topping Sauce])))

(define sauce-pizza% : Sauce-Pizza%
  (class object%
    (super-new)
    (field [topping 'tomato])))

(define (get-sauce [pizza : (Instance Pizza%)]) : Sauce
  (cond
    [(is-a? pizza sauce-pizza%)
     (get-field topping pizza)] ; type error
    [else
     'bbq]))
```

The type-error message explains that `(get-field topping pizza)` can return any kind of value, even when `pizza` is an instance of the `sauce-pizza%` class. In particular, `pizza` could be an instance of an untyped subclass that sets its `topping` to the integer `0`:

```
; #lang racket
(define evil-pizza%
  (class sauce-pizza%
    (inherit-field topping)
    (super-new)
    (set! topping 0)))
```

To downcast as intended, add a `cast` after the `is-a?` test. Below is a complete example that passes the type checker and raises a run-time error to prevent the typed `get-sauce` function from returning a non-`Sauce` value.

Examples:

```
> (module pizza typed/racket
    (provide get-sauce sauce-pizza%)

    (define-type Pizza%
      (Class (field [topping Any])))

    (define-type Sauce
      (U 'tomato 'bbq 'no-sauce))

    (define-type Sauce-Pizza%
      (Class #:implements Pizza% (field [topping Sauce])))

    (define sauce-pizza% : Sauce-Pizza%
      (class object%
        (super-new)
        (field [topping 'tomato])))

    (define (get-sauce [pizza : (Instance Pizza%)]) : Sauce
      (cond
        [(is-a? pizza sauce-pizza%)
         (define p+ (cast pizza (Instance Sauce-Pizza%)))
         (get-field topping p+)]
        [else
         'no-sauce])))
> (require 'pizza)
> (define evil-pizza%
    (class sauce-pizza%
      (inherit-field topping)
      (super-new)
```

```
      (set! topping 0)))
> (get-sauce (new evil-pizza%))
sauce-pizza%: broke its own contract
  promised: (or/c (quote no-sauce) (quote bbq) (quote
tomato))
  produced: 0
  in: the topping field in
      (recursive-contract g40 #:impersonator)
  contract from: pizza
  blaming: pizza
   (assuming the contract is correct)
  at: eval:1:0
```