The Typed Racket Guide

Version 5.1.2

Sam Tobin-Hochstadt <samth@racket-lang.org>
and Vincent St-Amour <stamourv@racket-lang.org>

August 3, 2011

Typed Racket is a family of languages, each of which enforce that programs written in the language obey a type system that ensures the absence of many common errors. This guide is intended for programmers familiar with Racket. For an introduction to Racket, see the ***The Racket Guide***.

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
(define-struct pt (x y))

; mag : pt -> number
(define (mag p)
  (sqrt (+ (sqr (pt-x p)) (sqr (pt-y p)))))
```

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

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

(: mag (pt -> Number))
(define (mag p)
  (sqrt (+ (sqr (pt-x p)) (sqr (pt-y p)))))
```

2 Beginning Typed Racket

Recall the typed module from §1 "Quick Start":

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

(: mag (pt -> Number))
(define (mag p)
  (sqrt (+ (sqr (pt-x p)) (sqr (pt-y p)))))
```

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]))
```

Many forms in Typed Racket have the same name as the untyped forms, with a `:` suffix.

This defines a new structure, name `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 the `struct` form from `racket`—when porting a program from `racket` to `typed/racket`, uses of `struct` should be changed to `struct:`.

```
(: mag (pt -> Number))
```

This declares that `mag` has the type `(pt -> Number)`.

The type `(pt -> Number)` is a function type, that is, the type of a procedure. The input type, or domain, is a single argument of type `pt`, which refers to an instance of the `pt` structure. The `->` both indicates that this is a function type and separates the domain from the range, or output type, in this case `Number`.

```
(define (mag p)
  (sqrt (+ (sqr (pt-x p)) (sqr (pt-y p)))))
```

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 `p` has the type `pt`, taking these types from the earlier type declaration. Since the body does have this type, the program is accepted.

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.

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:2:0: Type Checker: Expected Number, but got String in:
(quote "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.

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` have counterparts which allow the specification of types. The `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`, but type annotations are 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)`. Functions defined by cases may also be annotated:

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

This function has the type `(case-lambda (-> Number) (Number -> Number))`.

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 especially useful for binding forms which do not have counterparts provided by Typed Racket, such as `let+`:

```
(let+ ([val #{x : Number} (+ 6 1)])
  (* x x))
```

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: Expected Number, but got String in:
(quote "not a 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)`.

3.3 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 not be 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
- : Zero
0
> -7
- : Negative-Fixnum
-7
> 14
- : Positive-Byte
14
```

```
> 3.2
- : Positive-Flonum
3.2
> 7.0+2.8i
- : Float-Complex
7.0+2.8i
```

Finally, any value is itself a type:

```
> (ann 23 : 23)
- : 23
23
```

4.2 Function Types

We have already seen some examples of function types. Function types are constructed using `->`, with the argument types before the arrow and the result type after. 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 : ((! False @ 0) | (False @ 0)) (0))
#<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 Index))
#<procedure>
```

4.3 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)
```

4.4 Recursive Types

Recursive types can 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 (Rec BT (U Number (Pair BT BT))))
```

The `Rec` type constructor specifies that the type `BT` refers to the whole binary tree type within the body of the `Rec` form.

4.5 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`.

4.6 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))`.

4.7 Polymorphism

Typed Racket offers abstraction over types as well as values.

4.7.1 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: None ())
(struct: (a) Some ([v : a]))

(define-type (Opt a) (U None (Some a)))
```

```
(: find (Number (Listof Number) -> (Opt Number)))
(define (find v l)
  (cond [(null? l) (None)]
        [(= v (car l)) (Some v)]
        [else (find v (cdr l))]))
```

The first `struct:` defines `None` to be a structure with no contents.

The second definition

```
(struct: (a) Some ([v : a]))
```

creates a parameterized type, `Some`, which is a structure with one element, whose type is that of the type argument to `Some`. 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 (Opt a) (U None (Some a)))
```

creates a parameterized type — `Opt` is a potential container for whatever type is supplied.

The `find` function takes a number `v` and list, and produces `(Some v)` when the number is found in the list, and `(None)` otherwise. Therefore, it produces a `(Opt Number)`, just as the annotation specified.

4.7.2 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.

4.8 Variable-Arity Functions: Programming with Rest Arguments

Typed Racket can handle some uses of rest arguments.

4.8.1 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.

4.8.2 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 Optimization in Typed Racket

Typed Racket provides a type-driven optimizer that rewrites well-typed programs to potentially make them faster. It should in no way make your programs slower or unsafe.

For general information on Racket performance and benchmarking, see §18 "Performance".

5.1 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
```

5.2 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.

5.2.1 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 `exact->inexact`. 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`). Note that both parts of a literal must be present and inexact for the literal to be of type `Float-Complex`; `0.0+1.0i` is of type `Float-Complex` but `0+1.0i` is not. To get the most of Typed Racket's optimizer, you should also favor rectangular coordinates over polar coordinates.

5.2.2 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 `Integer`s 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))))
```

5.2.3 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.

```
; #(name 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.

5.2.4 Performance Debugging

Typed Racket provides performance debugging support to help you get the most of its optimizer.

Setting the racket logging facilities to the 'warning level when compiling a Typed Racket program causes the optimizer to log each optimization it performs. Setting the Racket logging level can be done on the command line with the -W flag:

```
racket -W warning my-typed-program.rkt
```

For example, the addition in the following program can be optimized:

```
#lang typed/racket
(: add-two-floats : Float Float -> Float)
(define (add-two-floats x y) (+ x y))
```

With optimizer logging turned on, the optimization is reported:

```
TR opt: #f 5:0 + -- binary float
```

In addition, the optimizer also reports cases where an optimization was close to happening, but was not ultimately safe to perform. Such missed optimization warnings usually provide explanations as to why an optimization could not be performed, pointing to changes to your program that would make it more amenable to optimization.

For example, the multiplication below cannot be safely optimized (see above discussion about mixing reals and floats), although it may look like it can.

```
#lang typed/racket
(: mul-int-float : Integer Float -> Real)
(define (mul-int-float x y) (* x y))
```

With optimizer logging turned on, the missed optimization is reported:

```
TR missed opt: #f 7:0 (#%app * x y) -- binary, args all float-arg-
expr, return type not Float -- caused by: 7:0 x
```