The Typed Racket Reference

Version 5.3.4

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

May 8, 2013

```
#lang typed/racket/base
#lang typed/racket
```

1 Type Reference

`Any`

Any Racket value. All other types are subtypes of `Any`.

`Nothing`

The empty type. No values inhabit this type, and any expression of this type will not evaluate to a value.

1.1 Base Types

1.1.1 Numeric Types

These types represent the hierarchy of numbers of Racket.

```
Number
Complex
```

`Number` and `Complex` are synonyms. This is the most general numeric type, including all Racket numbers, both exact and inexact, including complex numbers.

`Integer`

Includes Racket's exact integers and corresponds to the `exact-integer?` predicate. This is the most general type that is still valid for indexing and other operations that require integral values.

```
Float
Flonum
```

Includes Racket's double-precision (default) floating-point numbers and corresponds to the `flonum?` predicate. This type excludes single-precision floating-point numbers.

`Single-Flonum`

Includes Racket's single-precision floating-point numbers and corresponds to the `single-flonum?` predicate. This type excludes double-precision floating-point numbers.

`Inexact-Real`

Includes all of Racket's floating-point numbers, both single- and double-precision.

```
Exact-Rational
```

Includes Racket's exact rationals, which include fractions and exact integers.

```
Real
```

Includes all of Racket's real numbers, which include both exact rationals and all floating-point numbers. This is the most general type for which comparisons (e.g. <) are defined.

```
Exact-Number
Float-Complex
Single-Flonum-Complex
Inexact-Complex
```

These types correspond to Racket's complex numbers.

The above types can be subdivided into more precise types if you want to enforce tighter constraints. Typed Racket provides types for the positive, negative, non-negative and non-positive subsets of the above types (where applicable).

```
Positive-Integer
Exact-Positive-Integer
Nonnegative-Integer
Exact-Nonnegative-Integer
Natural
Negative-Integer
Nonpositive-Integer
Zero
Positive-Float
Positive-Flonum
Nonnegative-Float
Nonnegative-Flonum
Negative-Float
Negative-Flonum
Nonpositive-Float
Nonpositive-Flonum
Float-Negative-Zero
Flonum-Negative-Zero
Float-Positive-Zero
Flonum-Positive-Zero
Float-Zero
Flonum-Zero
Float-Nan
```

```
Flonum-Nan
Positive-Single-Flonum
Nonnegative-Single-Flonum
Negative-Single-Flonum
Nonpositive-Single-Flonum
Single-Flonum-Negative-Zero
Single-Flonum-Positive-Zero
Single-Flonum-Zero
Single-Flonum-Nan
Positive-Inexact-Real
Nonnegative-Inexact-Real
Negative-Inexact-Real
Nonpositive-Inexact-Real
Inexact-Real-Negative-Zero
Inexact-Real-Positive-Zero
Inexact-Real-Zero
Inexact-Real-Nan
Positive-Exact-Rational
Nonnegative-Exact-Rational
Negative-Exact-Rational
Nonpositive-Exact-Rational
Positive-Real
Nonnegative-Real
Negative-Real
Nonpositive-Real
Real-Zero
```

`Natural` and `Exact-Nonnegative-Integer` are synonyms. So are the integer and exact-integer types, and the float and flonum types. `Zero` includes only the integer `0`. `Real-Zero` includes exact `0` and all the floating-point zeroes.

These types are useful when enforcing that values have a specific sign. However, programs using them may require additional dynamic checks when the type-checker cannot guarantee that the sign constraints will be respected.

In addition to being divided by sign, integers are further subdivided into range-bounded types.

```
One
Byte
Positive-Byte
Index
Positive-Index
Fixnum
Positive-Fixnum
```

```
Nonnegative-Fixnum
Negative-Fixnum
Nonpositive-Fixnum
```

`One` includes only the integer `1`. `Byte` includes numbers from `0` to `255`. `Index` is bounded by `0` and by the length of the longest possible Racket vector. `Fixnum` includes all numbers represented by Racket as machine integers. For the latter two families, the sets of values included in the types are architecture-dependent, but typechecking is architecture-independent.

These types are useful to enforce bounds on numeric values, but given the limited amount of closure properties these types offer, dynamic checks may be needed to check the desired bounds at runtime.

Examples:

```
> 7
- : Integer [generalized from Positive-Byte]
7
> 8.3
- : Flonum [generalized from Positive-Flonum]
8.3
> (/ 8 3)
- : Exact-Rational [generalized from Positive-Exact-Rational]
8/3
> 0
- : Integer [generalized from Zero]
0
> -12
- : Integer [generalized from Negative-Fixnum]
-12
> 3+4i
- : Exact-Number
3+4i
```

1.1.2 Other Base Types

```
Boolean
True
False
String
Keyword
Symbol
Char
Void
```

```
Input-Port
Output-Port
Port
Path
Path-For-Some-System
Regexp
PRegexp
Byte-Regexp
Byte-PRegexp
Bytes
Namespace
Namespace-Anchor
Variable-Reference
Null
EOF
Continuation-Mark-Set
Undefined
Module-Path
Module-Path-Index
Resolved-Module-Path
Compiled-Module-Expression
Compiled-Expression
Internal-Definition-Context
Pretty-Print-Style-Table
Special-Comment
Struct-Type-Property
Impersonator-Property
Read-Table
Bytes-Converter
Parameterization
Custodian
Inspector
Security-Guard
UDP-Socket
TCP-Listener
Logger
Log-Receiver
Log-Level
Thread
Thread-Group
Subprocess
Place
Place-Channel
Semaphore
Will-Executor
```

```
Pseudo-Random-Generator
```

These types represent primitive Racket data.

Examples:

```
> #t
- : Boolean [generalized from True]
#t
> #f
- : False
#f
> "hello"
- : String
"hello"
> (current-input-port)
- : Input-Port
#<input-port:string>
> (current-output-port)
- : Output-Port
#<output-port:string>
> (string->path "/")
- : Path
#<path:/>
> #rx"a*b*"
- : Regexp
#rx"a*b*"
> #px"a*b*"
- : PRegexp
#px"a*b*"
> '#"bytes"
- : Bytes
#"bytes"
> (current-namespace)
- : Namespace
#<namespace:0>
> #\b
- : Char
#\b
> (thread (lambda () (add1 7)))
- : Thread
#<thread>
```

```
Path-String
```

The union of the `Path` and `String` types. Note that this does not match exactly what the predicate `path-string?` recognizes. For example, strings that contain the character `#\nul` have the type `Path-String` but `path-string?` returns `#f` for those strings. For a complete specification of which strings `path-string?` accepts, see its documentation.

1.2 Singleton Types

Some kinds of data are given singleton types by default. In particular, booleans, symbols, and keywords have types which consist only of the particular boolean, symbol, or keyword. These types are subtypes of `Boolean`, `Symbol` and `Keyword`, respectively.

Examples:

```
> #t
- : Boolean [generalized from True]
#t
> '#:foo
- : #:foo
'#:foo
> 'bar
- : Symbol [generalized from 'bar]
'bar
```

1.3 Containers

The following base types are parameteric in their type arguments.

```
(Pairof s t)
```

is the pair containing *s* as the `car` and *t* as the `cdr`

Examples:

```
> (cons 1 2)
- : (Pairof One Positive-Byte)
'(1 . 2)
> (cons 1 "one")
- : (Pairof One String)
'(1 . "one")
```

```
(Listof t)
```

Homogenous lists of *t*

```
(List t ...)
```

is the type of the list with one element, in order, for each type provided to the List type constructor.

```
(List t ... trest ... bound)
```

is the type of a list with one element for each of the *t*s, plus a sequence of elements corresponding to *trest*, where *bound* must be an identifier denoting a type variable bound with ....

```
(List* t t1 ... s)
```

is equivalent to (Pairof *t* (List* *t1* ... *s*)).

Examples:

```
> (list 'a 'b 'c)
- : (Listof (U 'a 'b 'c)) [generalized from (List 'a 'b 'c)]
'(a b c)
> (plambda: (a ...) ([sym : Symbol] boxes : (Boxof a) ... a)
    (ann (cons sym boxes) (List Symbol (Boxof a) ... a)))
- : (All (a ...) (Symbol (Boxof a) ... a -> (Pairof Symbol (List
(Boxof a) ... a))))
#<procedure>
> (map symbol->string (list 'a 'b 'c))
- : (Listof String) [generalized from (Pairof String (Listof
String))]
'("a" "b" "c")
```

```
(MListof t)
```

Homogenous mutable lists of *t*.

```
(MPairof t u)
```

Mutable pairs of *t* and *u*.

```
(Boxof t)
```

A box of *t*

Example:

```
> (box "hello world")
- : (Boxof String)
'#&"hello world"
```

```
(Vectorof t)
```

Homogenous vectors of *t*

```
(Vector t ...)
```

is the type of the list with one element, in order, for each type provided to the `Vector` type constructor.

```
FlVector
```

An flvector.

Examples:

```
> (vector 1 2 3)
- : (Vector Integer Integer Integer)
'#(1 2 3)
> #(a b c)
- : (Vector Symbol Symbol Symbol)
'#(a b c)
```

```
(HashTable k v)
```

is the type of a hash table with key type *k* and value type *v*.

Example:

```
> #hash((a . 1) (b . 2))
- : (HashTable Symbol Integer)
'#hash((b . 2) (a . 1))
```

```
(Setof t)
```

is the type of a set of *t*.

Example:

```
> (set 0 1 2 3)
- : (Setof Byte)
(set 0 1 2 3)
```

```
(Channelof t)
```

A channel on which only *t* s can be sent.

Example:

```
> (ann (make-channel) (Channelof Symbol))
- : (Channelof Symbol)
#<channel>
```

(Parameterof *t*)
(Parameterof *s* *t*)

A parameter of *t*. If two type arguments are supplied, the first is the type the parameter accepts, and the second is the type returned.

Examples:

```
> current-input-port
- : (Parameterof Input-Port)
#<procedure:current-input-port>
> current-directory
- : (Parameterof Path-String Path)
#<procedure:current-directory>
```

(Promise *t*)

A promise of *t*.

Example:

```
> (delay 3)
- : (Promise Positive-Byte)
#<promise:eval:36:0>
```

(Futureof *t*)

A future which produce a value of type *t* when touched.

(Sequenceof *t*)

A sequence that produces values of type *t* on each iteration.

(Custodian-Boxof *t*)

A custodian box of *t*.

(Thread-Cellof *t*)

A thread cell of *t*.

(Ephemeronof *t*)

An ephemeron whose value is of type *t*.

1.4 Syntax Objects

The following types represent syntax objects and their content.

`(Syntaxof t)`

A syntax object with content of type *t*. Applying `syntax-e` to a value of type `(Syntaxof t)` produces a value of type *t*.

`Identifier`

A syntax object containing a symbol. Equivalent to `(Syntaxof Symbol)`.

`Syntax`

A syntax object containing only symbols, keywords, strings, characters, booleans, numbers, boxes containing `Syntax`, vectors of `Syntax`, or (possibly improper) lists of `Syntax`. Equivalent to `(Syntaxof Syntax-E)`.

`Syntax-E`

The content of syntax objects of type `Syntax`. Applying `syntax-e` to a value of type `Syntax` produces a value of type `Syntax-E`.

`(Sexpof t)`

The recursive union of *t* with symbols, keywords, strings, characters, booleans, numbers, boxes, vectors, and (possibly improper) lists.

`Sexp`

Applying `syntax->datum` to a value of type `Syntax` produces a value of type `Sexp`. Equivalent to `(Sexpof Nothing)`.

`Datum`

Applying `datum->syntax` to a value of type `Datum` produces a value of type `Syntax`. Equivalent to `(Sexpof Syntax)`.

1.5 Control

The following types represent prompt tags and keys for continuation marks for use with delimited continuation functions and continuation mark functions.

```
(Prompt-Tagof s t)
```

A prompt tag to be used in a continuation prompt whose body produces the type *s* and whose handler has the type *t*. The type *t* must be a function type.

The domain of *t* determines the type of the values that can be aborted, using `abort-current-continuation`, to a prompt with this prompt tag.

Example:

```
> (make-continuation-prompt-tag 'prompt-tag)
- : (Prompt-Tagof Integer Integer)
#<continuation-prompt-tag:prompt-tag>
```

```
(Continuation-Mark-Keyof t)
```

A continuation mark key that is used for continuation mark operations such as `with-continuation-mark` and `continuation-mark-set->list`. The type *t* represents the type of the data that is stored in the continuation mark with this key.

Example:

```
> (make-continuation-mark-key 'mark-key)
- : (Continuation-Mark-Keyof Integer)
#<continuation-mark-key>
```

1.6 Other Type Constructors

```
(dom ... -> rng)
(dom ... rest * -> rng)
(dom ... rest ... bound -> rng)
(dom -> rng : pred)

dom = type
    | keyword type
    | [keyword type]
```

is the type of functions from the (possibly-empty) sequence *dom* `...` to the *rng* type. The second form specifies a uniform rest argument of type *rest*, and the third form specifies a non-uniform rest argument of type *rest* with bound *bound*. In the third form, the second occurrence of `...` is literal, and *bound* must be an identifier denoting a type variable. In the fourth form, there must be only one *dom* and *pred* is the type checked by the predicate. *dom* can include both mandatory and optional keyword arguments.

Examples:

```
> (λ: ([x : Number]) x)
- : (Number -> Number : ((! False @ 0) | (False @ 0)) (0))
#<procedure>
> (λ: ([x : Number]    y : String *)  (length y))
- : (Number String * -> Index)
#<procedure>
> ormap
- : (All (a c b ...) ((a b ... b -> c) (Listof a) (Listof b) ... b
-> (U False c)))
#<procedure:ormap>
> string?
- : (Any -> Boolean : String)
#<procedure:string?>
> (:print-type file->string)
(Path-String [#:mode (U 'binary 'text)] -> String)

> (: is-zero? : Number #:equality (Number Number -> Any) [#:zero Number] -> Any)

> (define (is-zero? n #:equality equality #:zero [zero 0])
    (equality n zero))

> (is-zero? 2 #:equality =)
- : Any
#f
> (is-zero? 2 #:equality eq? #:zero 2.0)
- : Any
#f
```

`Procedure`

is the supertype of all function types.

`(U t ...)`

is the union of the types *t* ....

Example:

```
> (λ: ([x : Real])(if (> 0 x) "yes" 'no))
- : (Real -> (U String 'no) : ((Real @ 0) | Bot))
#<procedure>
```

`(case-> fun-ty ...)`

is a function that behaves like all of the *fun-ty*s, considered in order from first to last. The *fun-ty*s must all be function types constructed with ->.

Example:

```
> (: add-map : (case->
                 [(Listof Integer) -> (Listof Integer)]
                 [(Listof Integer) (Listof Integer) -> (Listof Integer)]))
```

For the definition of add-map look into case-lambda:.

```
(t t1 t2 ...)
```

is the instantiation of the parametric type *t* at types *t1 t2* ...

```
(All (v ...) t)
```

is a parameterization of type *t*, with type variables *v* .... If *t* is a function type constructed with ->, the outer pair of parentheses around the function type may be omitted.

Examples:

```
> (: list-length : (All (A) (Listof A) -> Natural))

> (define (list-length lst)
    (if (null? lst)
        0
        (add1 (list-length (cdr lst)))))

> (list-length (list 1 2 3))
- : Integer [generalized from Nonnegative-Integer]
3
```

```
(Values t ...)
```

is the type of a sequence of multiple values, with types *t* .... This can only appear as the return type of a function.

Example:

```
> (values 1 2 3)
- : (Values Integer Integer Integer) [generalized from (Values One
Positive-Byte Positive-Byte)]
1
2
3
```

`v`

where `v` is a number, boolean or string, is the singleton type containing only that value

`(quote val)`

where `val` is a Racket value, is the singleton type containing only that value

`i`

where `i` is an identifier can be a reference to a type name or a type variable

`(Rec n t)`

is a recursive type where `n` is bound to the recursive type in the body `t`

Examples:

```
> (define-type IntList (Rec List (Pair Integer (U List Null))))

> (define-type (List A) (Rec List (Pair A (U List Null))))
```

`(Struct st)`

is a type which is a supertype of all instances of the potentially-polymorphic structure type `st`. Note that structure accessors for `st` will *not* accept `(Struct st)` as an argument.

→

An alias for `->`.

`case→`

An alias for `case->`.

∀

An alias for `All`.

1.7 Other Types

```
(Option t)
```

Either `t` or `#f`

```
(Opaque t)
```

A type constructed using `require-opaque-type`.

2 Special Form Reference

Typed Racket provides a variety of special forms above and beyond those in Racket. They are used for annotating variables with types, creating new types, and annotating expressions.

2.1 Binding Forms

loop, *f*, *a*, and *v* are names, *t* is a type. *e* is an expression and *body* is a block.

```
(let: ([v : t e] ...) . body)
(let: loop : t0 ([v : t e] ...) . body)
```

Local bindings, like `let`, each with associated types. In the second form, *t0* is the type of the result of *loop* (and thus the result of the entire expression as well as the final expression in *body*). Type annotations are optional.

Examples:

```
> (: filter-even : (Listof Natural) (Listof Natural) -> (Listof Natural))

> (define (filter-even lst accum)
    (if (null? lst)
        accum
        (let: ([first : Natural (car lst)]
               [rest  : (Listof Natural) (cdr lst)])
              (if (even? first)
                  (filter-even rest (cons first accum))
                  (filter-even rest accum)))))

> (filter-even (list 1 2 3 4 5 6) null)
- : (Listof Nonnegative-Integer)
'(6 4 2)
```

Examples:

```
> (: filter-even-loop : (Listof Natural) -> (Listof Natural))

> (define (filter-even-loop lst)
    (let: loop : (Listof Natural)
          ([accum : (Listof Natural) null]
           [lst   : (Listof Natural) lst])
          (cond
            [(null? lst)       accum]
            [(even? (car lst)) (loop (cons (car lst) accum) (cdr lst))]
            [else              (loop accum (cdr lst))])))
```

```
> (filter-even-loop (list 1 2 3 4))
- : (Listof Nonnegative-Integer)
'(4 2)
```

```
(plet: (a ...) ([v : t e] ...) . body)
```

A polymorphic version of `let:`, abstracted over the type variables `a`. The type variables `a` are bound in both the types of the formal, and in any type expressions in the `body`. Does not support the looping form of let.

```
(letrec: ([v : t e] ...) . body)
(let*: ([v : t e] ...) . body)
(let-values: ([([v : t] ...) e] ...) . body)
(letrec-values: ([([v : t] ...) e] ...) . body)
(let*-values: ([([v : t] ...) e] ...) . body)
```

Type-annotated versions of `letrec`, `let*`, `let-values`, `letrec-values`, and `let*-values`. As with `let:`, type annotations are optional.

```
(let/cc: v : t . body)
(let/ec: v : t . body)
```

Type-annotated versions of `let/cc` and `let/ec`.

2.2 Anonymous Functions

```
(lambda: formals . body)

formals = ([v : t] ...)
        | ([v : t] ...    v : t *)
        | ([v : t] ...    v : t ...)
```

A function of the formal arguments `v`, where each formal argument has the associated type. If a rest argument is present, then it has type `(Listof t)`.

```
(λ: formals . body)
```

An alias for the same form using `lambda:`.

```
(plambda: (a ...) formals . body)
```

A polymorphic function, abstracted over the type variables a. The type variables a are bound in both the types of the formal, and in any type expressions in the *body*.

```
(case-lambda: [formals body] ...)
```

A function of multiple arities. Note that each *formals* must have a different arity.

Example:

```
> (define add-map
    (case-lambda:
     [([lst : (Listof Integer)])
      (map add1 lst)]
     [([lst1 : (Listof Integer)]
       [lst2 : (Listof Integer)])
      (map + lst1 lst2)]))
```

For the type declaration of add-map look at case-lambda.

```
(pcase-lambda: (a ...) [formals body] ...)
```

A polymorphic function of multiple arities.

```
(opt-lambda: formals . body)

formals = ([v : t] ... [v : t default] ...)
        | ([v : t] ... [v : t default] ...    v : t *)
        | ([v : t] ... [v : t default] ...    v : t ...)
```

A function with optional arguments.

```
(popt-lambda: (a ...) formals . body)
```

A polymorphic function with optional arguments.

2.3 Loops

```
(for: type-ann-maybe (for:-clause ...)
  expr ...+)
```

```
type-ann-maybe =
               | : u

   for:-clause = [id : t seq-expr]
               | [id seq-expr]
               | #:when guard
```

Like for, but each *id* having the associated type *t*. Since the return type is always Void, annotating the return type of a for form is optional. Unlike for, multi-valued *seq-expr*s are not supported. Type annotations in clauses are optional for all for: variants.

```
(for/list: type-ann-maybe (for:-clause ...) expr ...+)
(for/hash: type-ann-maybe (for:-clause ...) expr ...+)
(for/hasheq: type-ann-maybe (for:-clause ...) expr ...+)
(for/hasheqv: type-ann-maybe (for:-clause ...) expr ...+)
(for/vector: type-ann-maybe (for:-clause ...) expr ...+)
(for/flvector: type-ann-maybe (for:-clause ...) expr ...+)
(for/and: type-ann-maybe (for:-clause ...) expr ...+)
(for/or:   type-ann-maybe (for:-clause ...) expr ...+)
(for/first: type-ann-maybe (for:-clause ...) expr ...+)
(for/last: type-ann-maybe (for:-clause ...) expr ...+)
(for/sum: type-ann-maybe (for:-clause ...) expr ...+)
(for/product: type-ann-maybe (for:-clause ...) expr ...+)
(for*/list: type-ann-maybe (for:-clause ...) expr ...+)
(for*/hash: type-ann-maybe (for:-clause ...) expr ...+)
(for*/hasheq: type-ann-maybe (for:-clause ...) expr ...+)
(for*/hasheqv: type-ann-maybe (for:-clause ...) expr ...+)
(for*/vector: type-ann-maybe (for:-clause ...) expr ...+)
(for*/flvector: type-ann-maybe (for:-clause ...) expr ...+)
(for*/and: type-ann-maybe (for:-clause ...) expr ...+)
(for*/or:   type-ann-maybe (for:-clause ...) expr ...+)
(for*/first: type-ann-maybe (for:-clause ...) expr ...+)
(for*/last: type-ann-maybe (for:-clause ...) expr ...+)
(for*/sum: type-ann-maybe (for:-clause ...) expr ...+)
(for*/product: type-ann-maybe (for:-clause ...) expr ...+)
```

These behave like their non-annotated counterparts, with the exception that #:when clauses can only appear as the last *for:-clause*. The return value of the entire form must be of type u. For example, a for/list: form would be annotated with a Listof type. All annotations are optional.

```
(for/lists: type-ann-maybe ([id : t] ...)
  (for:-clause ...)
  expr ...+)
(for/fold:  type-ann-maybe ([id : t init-expr] ...)
  (for:-clause ...)
  expr ...+)
```

These behave like their non-annotated counterparts. Unlike the above, #:when clauses can be used freely with these.

```
(for*: void-ann-maybe (for-clause ...)
  expr ...+)
(for*/lists: type-ann-maybe ([id : t] ...)
  (for:-clause ...)
  expr ...+)
(for*/fold:  type-ann-maybe ([id : t init-expr] ...)
  (for:-clause ...)
  expr ...+)
```

These behave like their non-annotated counterparts.

```
for
for*
```

These are identical to for and for*, but provide additional annotations to help the typechecker.

```
(do: : u ([id : t init-expr step-expr-maybe] ...)
         (stop?-expr finish-expr ...)
  expr ...+)

step-expr-maybe =
                 | step-expr
```

Like do, but each *id* having the associated type *t*, and the final body *expr* having the type *u*. Type annotations are optional.

2.4 Definitions

```
(define: v : t e)
(define: (f . formals) : t . body)
(define: (a ...) v : t e)
(define: (a ...) (f . formals) : t . body)
```

These forms define variables, with annotated types. The first form defines *v* with type *t* and value *e*. The third form does the same, but allows the specification of type variables. The second and fourth forms defines a function *f* with appropriate types. In most cases, use of *:* is preferred to use of define:.

Examples:

```
> (define: foo : Integer 10)

> (define: (add [first : Integer]
                [rest  : Integer]) : Integer
    (+ first rest))

> (define: (A) mt-seq : (Sequenceof A) empty-sequence)

> (define: (A) (poly-app [func : (A A -> A)]
                         [first : A]
                         [rest  : A]) : A
    (func first rest))
```

2.5 Structure Definitions

```
(struct: maybe-type-vars name-spec ([f : t] ...) options ...)

maybe-type-vars =
                | (v ...)

      name-spec = name
                | name parent

        options = #:transparent
                | #:mutable
```

Defines a structure with the name *name*, where the fields *f* have types *t*, similar to the behavior of `struct`. When *parent* is present, the structure is a substructure of *parent*. When *maybe-type-vars* is present, the structure is polymorphic in the type variables *v*. If *parent* is also a polymorphic struct, then there must be at least as many type variables as in the parent type, and the parent type is instantiated with a prefix of the type variables matching the amount it needs.

Options provided have the same meaning as for the `struct` form.

```
(define-struct: maybe-type-vars name-spec ([f : t] ...) options ...)

maybe-type-vars =
                | (v ...)

      name-spec = name
                | (name parent)

        options = #:transparent
                | #:mutable
```

Legacy version of `struct:`, corresponding to `define-struct`.

```
(define-struct/exec: name-spec ([f : t] ...) [e : proc-t])

name-spec = name
          | (name parent)
```

Like `define-struct:`, but defines a procedural structure. The procdure *e* is used as the value for `prop:procedure`, and must have type *proc-t*.

2.6 Names for Types

```
(define-type name t)
(define-type (name v ...) t)
```

The first form defines *name* as type, with the same meaning as *t*. The second form is equivalent to `(define-type name (All (v ...) t))`. Type names may refer to other types defined in the same module, but cycles among them are prohibited.

Examples:

```
> (define-type IntStr (U Integer String))

> (define-type (ListofPairs A) (Listof (Pair A A)))
```

2.7 Generating Predicates Automatically

```
(make-predicate t)
```

Evaluates to a predicate for the type *t*, with the type `(Any -> Boolean : t)`. *t* may not contain function types, or types that may refer to mutable data such as `(Vectorof Integer)`.

```
(define-predicate name t)
```

Equivalent to `(define name (make-predicate t))`.

2.8 Type Annotation and Instantiation

```
(: v t)
```

This declares that *v* has type *t*. The definition of *v* must appear after this declaration. This can be used anywhere a definition form may be used.

Examples:

```
> (: var1 Integer)

> (: var2 String)
```

```
(provide: [v t] ...)
```

This declares that the *v*s have the types *t*, and also provides all of the *v*s.

```
#{v : t}
```

This declares that the variable *v* has type *t*. This is legal only for binding occurrences of *v*.

```
(ann e t)
```

Ensure that *e* has type *t*, or some subtype. The entire expression has type *t*. This is legal only in expression contexts.

```
#{e :: t}
```

A reader abbreviation for `(ann e t)`.

```
(cast e t)
```

The entire expression has the type *t*, while *e* may have any type. The value of the entire expression is the value returned by *e*, protected by a contract ensuring that it has type *t*. This is legal only in expression contexts.

Examples:

```
> (cast 3 Integer)
- : Integer
3
> (cast 3 String)
3: broke its contract
 promised: String
 produced: 3
 in: String
 contract from: cast
 blaming: cast
 at: eval:18.0
> (cast (lambda: ([x : Any]) x) (String -> String))
- : (String -> String)
#<procedure:val>
```

```
(inst e t ...)
```

Instantiate the type of `e` with types `t ....` `e` must have a polymorphic type with the appropriate number of type variables. This is legal only in expression contexts.

Examples:

```
> (foldl (inst cons Integer Integer) null (list 1 2 3 4))
- : (Listof Integer)
'(4 3 2 1)
> (: fold-list : (All (A) (Listof A) -> (Listof A)))

> (define (fold-list lst)
    (foldl (inst cons A A) null lst))

> (fold-list (list "1" "2" "3" "4"))
- : (Listof String)
'("4" "3" "2" "1")
```

```
#{e @ t ...}
```

A reader abbreviation for `(inst e t ...)`.

2.9 Require

Here, *m* is a module spec, *pred* is an identifier naming a predicate, and *r* is an optionally-renamed identifier.

```
(require/typed m rt-clause ...)

    rt-clause = [r t]
              | [#:struct name ([f : t] ...)
                     struct-option ...]
              | [#:struct (name parent) ([f : t] ...)
                     struct-option ...]
              | [#:opaque t pred]

struct-option = #:constructor-name constructor-id
              | #:extra-constructor-name constructor-id
```

This form requires identifiers from the module `m`, giving them the specified types.

The first case requires `r`, giving it type `t`.

The second and third cases require the struct with name *name* with fields `f ...`, where each field has type *t*. The third case allows a `parent` structure type to be specified. The parent type must already be a structure type known to Typed Racket, either built-in or via `require/typed`. The structure predicate has the appropriate Typed Racket filter type so that it may be used as a predicate in `if` expressions in Typed Racket.

Examples:

```
> (module UNTYPED racket/base
    (define n 100)

    (struct IntTree
      (elem left right))

    (provide n (struct-out IntTree)))

> (module TYPED typed/racket
    (require/typed 'UNTYPED
                   [n Natural]
                   [#:struct IntTree
                     ([elem  : Integer]
                      [left  : IntTree]
                      [right : IntTree])]))
```

The fourth case defines a new type *t*. `pred`, imported from module `m`, is a predicate for this type. The type is defined as precisely those values to which `pred` produces `#t`. `pred` must have type `(Any -> Boolean)`. Opaque types must be required lexically before they are used.

In all cases, the identifiers are protected with contracts which enforce the specified types. If this contract fails, the module `m` is blamed.

Some types, notably the types of predicates such as `number?`, cannot be converted to contracts and raise a static error when used in a `require/typed` form. Here is an example of using `case->` in `require/typed`.

```
(require/typed racket/base
               [file-or-directory-modify-seconds
                (case->
                  [String -> Exact-Nonnegative-Integer]
                  [String (Option Exact-Nonnegative-Integer)
                          ->
                          (U Exact-Nonnegative-Integer Void)]
                  [String (Option Exact-Nonnegative-
Integer) (-> Any)
                          ->
                          Any])])
```

file-or-directory-modify-seconds has some arguments which are optional, so we need to use case->.

```
(require/typed/provide m rt-clause ...)
```

Similar to require/typed, but also provides the imported identifiers.

2.10 Other Forms

```
with-handlers
lambda
λ
define
```

Identical to with-handlers, lambda, λ, and define, respectively, but provide additional annotations to assist the typechecker. The define:, lambda:, and λ: forms are useful replacements which support type annotation.

Note that unlike define, define does not bind functions with keyword arguments to static information about those functions.

```
(default-continuation-prompt-tag)
 → (-> (Prompt-Tagof Any (Any -> Any)))
```

Identical to default-continuation-prompt-tag, but additionally protects the resulting prompt tag with a contract that wraps higher-order values, such as functions, that are communicated with that prompt tag. If the wrapped value is used in untyped code, a contract error will be raised.

Examples:

```
> (module typed typed/racket
    (provide do-abort)
    (: do-abort (-> Void))
    (define (do-abort)
      (abort-current-continuation
       ; typed, and thus contracted, prompt tag
       (default-continuation-prompt-tag)
       (λ: ([x : Integer]) (+ 1 x)))))

> (module untyped racket
    (require 'typed)
    (call-with-continuation-prompt
      (λ () (do-abort))
```

```
    (default-continuation-prompt-tag)
    ; the function cannot be passed an argument
    (λ (f) (f 3))))
```

```
> (require 'untyped)
```

default-continuation-prompt-tag: broke its contract
Attempted to use a higher-order value passed as ‘Any‘ in untyped code: #<procedure>
in: ...
the range of
(-> (prompt-tag/c Any #:call/cc Any))
contract from: untyped
blaming: untyped

`(#%module-begin form ...)`

Legal only in a module begin context. The `#%module-begin` form of `typed/racket` checks all the forms in the module, using the Typed Racket type checking rules. All `provide` forms are rewritten to insert contracts where appropriate. Otherwise, the `#%module-begin` form of `typed/racket` behaves like `#%module-begin` from `racket`.

`(#%top-interaction . form)`

Performs type checking of forms entered at the read-eval-print loop. The `#%top-interaction` form also prints the type of *form* after type checking.

3 Libraries Provided With Typed Racket

The `typed/racket` language corresponds to the `racket` language—that is, any identifier provided by `racket`, such as `modulo` is available by default in `typed/racket`.

```
#lang typed/racket
(modulo 12 2)
```

The `typed/racket/base` language corresponds to the `racket/base` language.

Some libraries have counterparts in the `typed` collection, which provide the same exports as the untyped versions. Such libraries include `srfi/14`, `net/url`, and many others.

```
#lang typed/racket
(require typed/srfi/14)
(char-set= (string->char-set "hello")
           (string->char-set "olleh"))
```

To participate in making more libraries available, please visit here.

Other libraries can be used with Typed Racket via `require/typed`.

```
#lang typed/racket
(require/typed version/check
               [check-version (-> (U Symbol (Listof Any)))])
(check-version)
```

4 Utilities

Typed Racket provides some additional utility functions to facilitate typed programming.

```
(assert v) → A
  v : (U #f A)
(assert v p?) → B
  v : A
  p? : (A -> Any : B)
```

Verifies that the argument satisfies the constraint. If no predicate is provided, simply checks that the value is not `#f`.

See also the `cast` form.

Examples:

```
> (define: x : (U #f String) (number->string 7))

> x
- : (U False String)
"7"
> (assert x)
"7"
> (define: y : (U String Symbol) "hello")

> y
- : (U Symbol String)
"hello"
> (assert y string?)
"hello"
> (assert y boolean?)
```

Assertion failed

```
(with-asserts ([id maybe-pred] ...) body ...+)

maybe-pred =
           | predicate
```

Guard the body with assertions. If any of the assertions fail, the program errors. These assertions behave like `assert`.

```
(defined? v) → boolean?
  v : any/c
```

A predicate for determining if `v` is *not* `#<undefined>`.

```
(index? v) → boolean?
  v : any/c
```

A predicate for the Index type.

```
(typecheck-fail orig-stx maybe-msg maybe-id)

maybe-msg =
          | msg-string

 maybe-id =
          | #:covered-id id
```

Explicitly produce a type error, with the source location or *orig-stx*. If *msg-string* is present, it must be a literal string, it is used as the error message, otherwise the error message "Incomplete case coverage" is used. If *id* is present and has type T, then the message "missing coverage of T" is added to the error message.

Examples:

```
> (define-syntax (cond* stx)
    (syntax-case stx ()
      [(_ x clause ...)
       #`(cond clause ... [else (typecheck-fail #,stx "incomplete
coverage"
                                               #:covered-
id x)])]))

> (define: (f [x  : (U String Integer)]) : Boolean
    (cond* x
           [(string? x) #t]
           [(exact-nonnegative-integer? x) #f]))
```

Type Checker: incomplete coverage; missing coverage of Negative-Integer
in: #%top-interaction

5 Exploring Types

In addition to printing a summary of the types of REPL results, Typed Racket provides interactive utilities to explore and query types. The following bindings are only available at the Typed Racket REPL.

```
(:type t)
```

Prints the type *t*.

```
(:print-type e)
```

Prints the type of *e*. This prints the whole type, which can sometimes be quite large.

```
(:query-type/args f t ...)
```

Given a function *f* and argument types *t*, shows the result type of *f*.

```
(:query-type/result f t)
```

Given a function *f* and a desired return type *t*, shows the arguments types *f* should be given to return a value of type *t*.

6 Typed Racket Syntax Without Type Checking

```
#lang typed/racket/no-check
#lang typed/racket/base/no-check
```

On occasions where the Typed Racket syntax is useful, but actual typechecking is not desired, the `typed/racket/no-check` and `typed/racket/base/no-check` languages are useful. They provide the same bindings and syntax as `typed/racket` and `typed/racket/base`, but do no type checking.

Examples:

```
#lang typed/racket/no-check
(: x Number)
(define x "not-a-number")
```

7 Typed Regions

The `with-type` form allows for localized Typed Racket regions in otherwise untyped code.

```
(with-type result-spec fv-clause body ...+)
(with-type export-spec fv-clause body ...+)

  fv-clause =
            | #:freevars ([id fv-type] ...)

result-spec = #:result type

export-spec = ([export-id export-type] ...)
```

The first form, an expression, checks that `body ...+` has the type `type`. If the last expression in `body ...+` returns multiple values, `type` must be a type of the form `(values t ...)`. Uses of the result values are appropriately checked by contracts generated from `type`.

The second form, which can be used as a definition, checks that each of the `export-id`s has the specified type. These types are also enforced in the surrounding code with contracts.

The `id`s are assumed to have the types ascribed to them; these types are converted to contracts and checked dynamically.

Examples:

```
> (with-type #:result Number 3)
3
> ((with-type #:result (Number -> Number)
     (lambda: ([x : Number]) (add1 x)))
   #f)
contract violation:
 expected: Number
 given: #f
 in: the 1st argument of
      (-> Number Number)
 contract from: (region typed-region)
 blaming: top-level
> (let ([x "hello"])
    (with-type #:result String
      #:freevars ([x String])
      (string-append x ", world")))
"hello, world"
> (let ([x 'hello])
    (with-type #:result String
      #:freevars ([x String])
```

```
      (string-append x ", world")))
```

x: broke its contract
promised: String
produced: 'hello
in: String
contract from: top-level
blaming: top-level
at: eval:5.0

```
> (with-type ([fun (Number -> Number)]
              [val Number])
    (define (fun x) x)
    (define val 17))

> (fun val)
17
```

8 Optimization in Typed Racket

[1]

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.

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
```

[1]See §5 "Optimization in Typed Racket" in the guide for tips to get the most out of the optimizer.

9 Legacy Forms

The following forms are provided by Typed Racket for backwards compatibility.

```
define-type-alias
```

Equivalent to `define-type`.

```
define-typed-struct
```

Equivalent to `define-struct:`

```
require/opaque-type
```

Similar to using the `opaque` keyword with `require/typed`.

```
require-typed-struct
```

Similar to using the `struct` keyword with `require/typed`.

```
require-typed-struct/provide
```

Similar to `require-typed-struct`, but also provides the imported identifiers.

```
pdefine:
```

Defines a polymorphic function.

```
(pred t)
```

Equivalent to `(Any -> Boolean : t)`.

```
Un
```

An alias for `U`.

```
mu
```

An alias for `Rec`.

`Tuple`

An alias for `List`.

`Parameter`

An alias for `Parameterof`.

`Pair`

An alias for `Pairof`.

`values`

An alias for `Values`.

10 Compatibility Languages

```
#lang typed/scheme
#lang typed/scheme/base
#lang typed-scheme
```

Typed versions of the

```
#lang scheme
```

and

```
#lang scheme/base
```

languages. The

```
#lang typed-scheme
```

language is equivalent to the

```
#lang typed/scheme/base
```

language.

```
(require/typed m rt-clause ...)

    rt-clause = [r t]
              | [struct name ([f : t] ...)
                     struct-option ...]
              | [struct (name parent) ([f : t] ...)
                     struct-option ...]
              | [opaque t pred]

struct-option = #:constructor-name constructor-id
              | #:extra-constructor-name constructor-id
```

Similar to `require/typed`, but as if `#:extra-constructor-name make-name` was supplied.

```
require-typed-struct
```

Similar to using the `struct` keyword with `require/typed`.

11 Experimental Features

These features are currently experimental and subject to change.

```
(Class args ...)
```

A type constructor for typing classes created using `racket/class`.

```
(Instance c)
```

A type constructor for typing objects created using `racket/class`.

```
(declare-refinement id)
```

Declares *id* to be usable in refinement types.

```
(Refinement id)
```

Includes values that have been tested with the predicate *id*, which must have been specified with `declare-refinement`.

```
(define-typed-struct/exec forms ...)
```

Defines an executable structure.