The Typed Racket Reference

Version 9.0.0.1

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 20, 2025

This manual describes the Typed Racket language, a sister language of Racket with a static type-checker. The types, special forms, and other tools provided by Typed Racket are documented here.

For a friendly introduction, see the companion manual *The Typed Racket Guide*. For technical details, refer to the "Bibliography".

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

package: `typed-racket-lib`

1 Type Reference

`Any`

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

`AnyValues`

Any number of Racket values of any type.

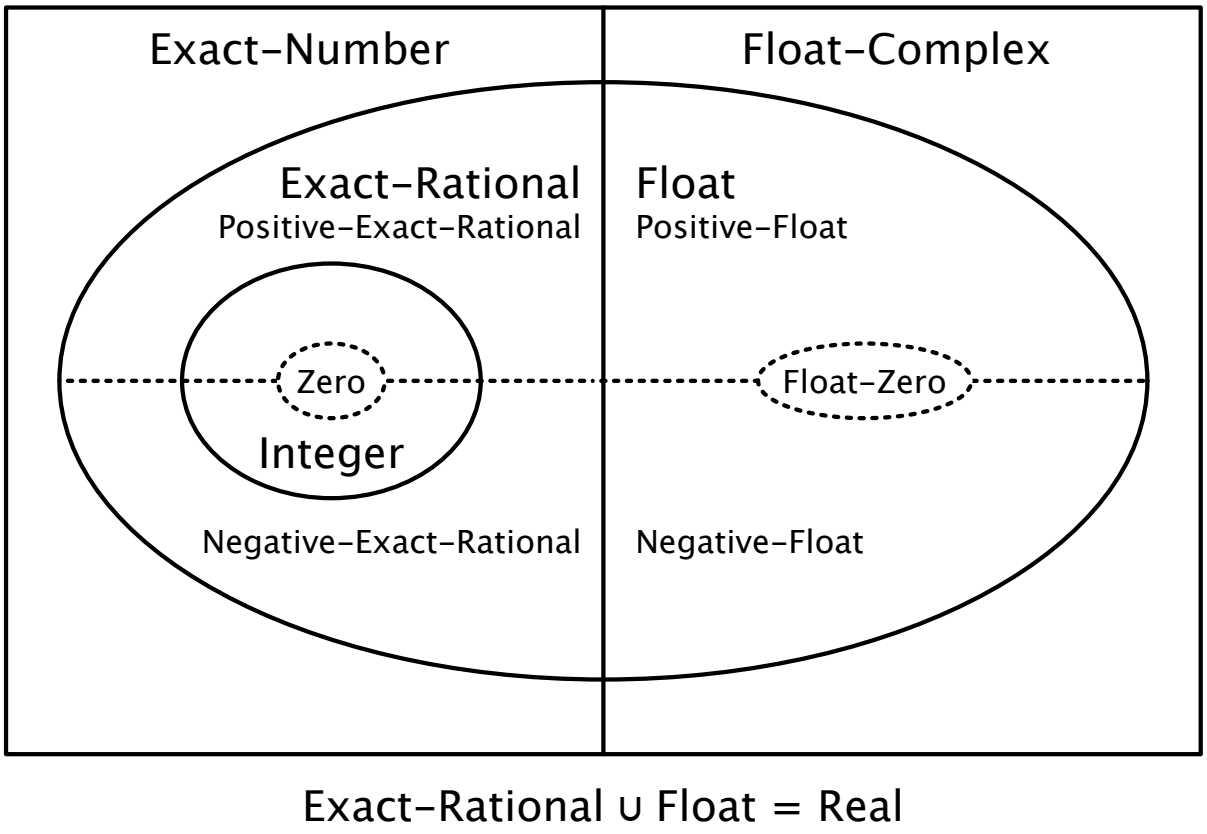
`Nothing`

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. The diagram below shows the relationships between the types in the hierarchy.



The regions with a solid border are *layers* of the numeric hierarchy corresponding to sets of numbers such as integers or rationals. Layers contained within another are subtypes of the layer containing them. For example, `Exact-Rational` is a subtype of `Exact-Number`.

The `Real` layer is also divided into positive and negative types (shown with a dotted line). The `Integer` layer is subdivided into several fixed-width integers types, detailed later in this section.

```
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
Imaginary
Exact-Complex
Exact-Imaginary
Inexact-Imaginary
```

These types correspond to Racket's complex numbers.

Changed in version 1.7 of package `typed-racket-lib`: Added `Imaginary`, `Inexact-Complex`, `Exact-Complex`, `Exact-Imaginary`, `Inexact-Imaginary`.

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).

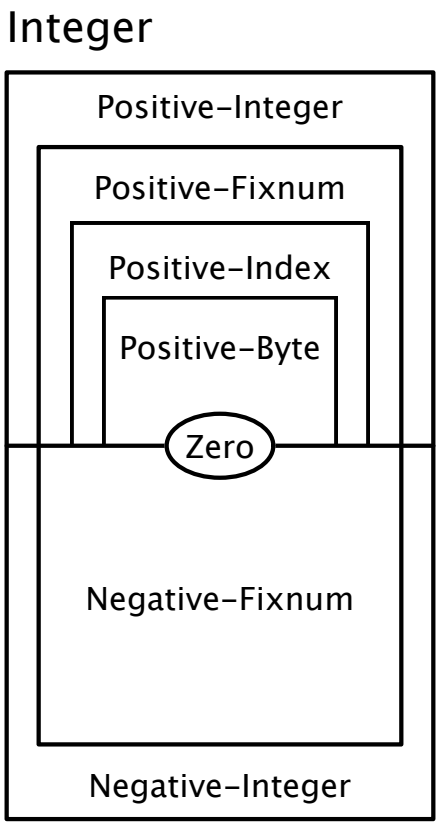
```
Positive-Integer
Exact-Positive-Integer
Nonnegative-Integer
Exact-Nonnegative-Integer
Natural
Negative-Integer
Nonpositive-Integer
Zero
Positive-Float
Positive-Flonum
Nonnegative-Float
Nonnegative-Flonum
Negative-Float
Negative-Flonum
Nonpositive-Float
Nonpositive-Flonum
Float-Negative-Zero
Flonum-Negative-Zero
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. The relationships between most of the range-bounded types are shown in this diagram:



Like the previous diagram, types nested inside of another in the diagram are subtypes of its

containing 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 [more precisely: Positive-Byte]
7
> 8.3
- : Flonum [more precisely: Positive-Float-No-NaN]
8.3
> (/ 8 3)
- : Exact-Rational [more precisely: Positive-Exact-Rational]
8/3
> 0
- : Integer [more precisely: Zero]
0
> -12
- : Integer [more precisely: Negative-Fixnum]
-12
> 3+4i
- : Exact-Number
3+4i
```

```
ExtFlonum
Positive-ExtFlonum
Nonnegative-ExtFlonum
```

```
Negative-ExtFlonum
Nonpositive-ExtFlonum
ExtFlonum-Negative-Zero
ExtFlonum-Positive-Zero
ExtFlonum-Zero
ExtFlonum-Nan
```

80-bit extflonum types, for the values operated on by `racket/extflonum` exports. These are not part of the numeric tower.

1.1.2 Other Base Types

```
Boolean
True
False
String
Keyword
Symbol
Char
Void
Input-Port
Output-Port
Unquoted-Printing-String
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
FSemaphore
Will-Executor
Pseudo-Random-Generator
Environment-Variables
```

These types represent primitive Racket data.

Examples:

```
> #t
- : 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>
> #\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
- : True
#t
> '#:foo
- : '#:foo
'#:foo
> 'bar
- : 'bar
'bar
```

1.3 Base Type Constructors and Supertypes

```
(Pairof s t)
```

Returns a pair type 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)
```

Returns the type of a homogeneous list of *t*

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

Returns a list type with one element, in order, for each type provided to the `List` type constructor.

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

Returns 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))`. `(List* s)` is equivalent to *s* itself.

Examples:

```
> (list 'a 'b 'c)
- : (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))
- : (Pairof String (Listof String))
'("a" "b" "c")
```

```
(MListof t)
```

Returns the type of a homogeneous mutable list of `t`.

```
(MPairof t u)
```

Returns the type of a Mutable pair of `t` and `u`.

```
(TreeListof t)
```

Returns the type of treelist of `t`

```
MPairTop
```

Is the type of a mutable pair with unknown element types and is the supertype of all mutable pair types. This type typically appears in programs via the combination of occurrence typing and `mpair?`.

Example:

```
> (lambda: ([x : Any]) (if (mpair? x) x (error "not an mpair!")))
- : (-> Any MPairTop)
#<procedure>
```

```
(Boxof t)
```

Returns the type of a box of `t`

Example:

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

```
BoxTop
```

Is the type of a box with an unknown element type and is the supertype of all box types. Only read-only box operations (e.g. `unbox`) are allowed on values of this type. This type typically appears in programs via the combination of occurrence typing and `box?`.

Example:

```
> (lambda: ([x : Any]) (if (box? x) x (error "not a box!")))
- : (-> Any BoxTop)
#<procedure>
```

`(Vectorof t)`

Returns the type of a homogeneous vector list of `t` (mutable or immutable).

`(Immutable-Vectorof t)`

Returns the type of a homogeneous immutable vector of `t`.

Added in version 1.9 of package `typed-racket-lib`.

`(Mutable-Vectorof t)`

Returns the type of a homogeneous mutable vector of `t`.

Added in version 1.9 of package `typed-racket-lib`.

`(Vector t ...)`

Returns the type of a mutable or immutable vector with one element, in order, for each type provided to the `Vector` type constructor.

Example:

```
> (ann (vector 1 'A) (Vector Fixnum 'A))
- : (U (Immutable-Vector Fixnum 'A) (Mutable-Vector Fixnum 'A))
'#(1 A)
```

`(Immutable-Vector t ...)`

Similar to `(Vector t ...)`, but for immutable vectors.

Example:

```
> (vector-immutable 1 2 3)
- : (Immutable-Vector One Positive-Byte Positive-Byte)
'#(1 2 3)
```

Added in version 1.9 of package `typed-racket-lib`.

`(Mutable-Vector t ...)`

Similar to `(Vector t ...)`, but for mutable vectors.

Example:

```
> (vector 1 2 3)
- : (Mutable-Vector Integer Integer Integer)
'#(1 2 3)
```

Added in version 1.9 of package `typed-racket-lib`.

`FlVector`

An flvector.

Example:

```
> (flvector 1.0 2.0 3.0)
- : FlVector
(flvector 1.0 2.0 3.0)
```

`ExtFlVector`

An extflvector.

Example:

```
> (extflvector 1.0t0 2.0t0 3.0t0)
- : ExtFlVector
#<extflvector>
```

`FxVector`

An fxvector.

Example:

```
> (fxvector 1 2 3)
- : FxVector
(fxvector 1 2 3)
```

VectorTop

Is the type of a vector with unknown length and element types and is the supertype of all vector types. Only read-only vector operations (e.g. `vector-ref`) are allowed on values of this type. This type typically appears in programs via the combination of occurrence typing and `vector?`.

Example:

```
> (lambda: ([x : Any]) (if (vector? x) x (error "not a vector!")))
- : (-> Any VectorTop)
#<procedure>
```

Mutable-VectorTop

Is the type of a mutable vector with unknown length and element types.

(HashTable *k* *v*)

Returns the type of a mutable or immutable hash table with key type *k* and value type *v*.

Example:

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

(Immutable-HashTable *k* *v*)

Returns the type of an immutable hash table with key type *k* and value type *v*.

Example:

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

Added in version 1.8 of package `typed-racket-lib`.

```
(Mutable-HashTable k v)
```

Returns the type of a mutable hash table that holds keys strongly (see §16.1 “Weak Boxes”) with key type *k* and value type *v*.

Example:

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

Added in version 1.8 of package `typed-racket-lib`.

```
(Weak-HashTable k v)
```

Returns the type of a mutable hash table that holds keys weakly with key type *k* and value type *v*.

Example:

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

Added in version 1.8 of package `typed-racket-lib`.

```
HashTableTop
```

Is the type of a hash table with unknown key and value types and is the supertype of all hash table types. Only read-only hash table operations (e.g. `hash-ref`) are allowed on values of this type. This type typically appears in programs via the combination of occurrence typing and `hash?`.

Example:

```
> (lambda: ([x : Any]) (if (hash? x) x (error "not a hash
table!")))
- : (-> Any HashTableTop)
#<procedure>
```

```
Mutable-HashTableTop
```

Is the type of a mutable hash table that holds keys strongly with unknown key and value types.

Weak-HashTableTop

Is the type of a mutable hash table that holds keys weakly with unknown key and value types.

(Setof *t*)

Returns the type of a hash set of *t*. This includes custom hash sets, but not mutable hash set or sets that are implemented using `gen:set`.

Example:

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

Example:

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

(Channelof *t*)

Returns the type of a channel on which only *t*s can be sent.

Example:

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

ChannelTop

Is the type of a channel with unknown message type and is the supertype of all channel types. This type typically appears in programs via the combination of occurrence typing and `channel?`.

Example:

```
> (lambda: ([x : Any]) (if (channel? x) x (error "not a
channel!")))
- : (-> Any ChannelTop)
#<procedure>
```

```
(Async-Channelof t)
```

Returns the type of an asynchronous channel on which only `t`s can be sent.

Examples:

```
> (require typed/racket/async-channel)
> (ann (make-async-channel) (Async-Channelof Symbol))
- : (Async-Channelof Symbol)
#<async-channel>
```

Added in version 1.1 of package `typed-racket-lib`.

```
Async-ChannelTop
```

Is the type of an asynchronous channel with unknown message type and is the supertype of all asynchronous channel types. This type typically appears in programs via the combination of occurrence typing and `async-channel?`.

Examples:

```
> (require typed/racket/async-channel)
> (lambda: ([x : Any]) (if (async-channel? x) x (error "not an
async-channel!")))
- : (-> Any Async-ChannelTop)
#<procedure>
```

Added in version 1.1 of package `typed-racket-lib`.

```
(Parameterof t)
(Parameterof s t)
```

Returns the type of 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)
```

Returns the type of promise of `t`.

Example:

```
> (delay 3)
- : (Promise Positive-Byte)
#<promise:eval:52:0>
```

```
(Futureof t)
```

Returns the type of future which produce a value of type `t` when touched.

```
(Sequenceof t ...)
```

Returns the type of sequence that produces `(Values t ...)` on each iteration. E.g., `(Sequenceof)` is a sequence which produces no values, `(Sequenceof String)` is a sequence of strings, `(Sequenceof Number String)` is a sequence which produces two values—a number and a string—on each iteration, etc.

```
SequenceTop
```

Is the type of a sequence with unknown element type and is the supertype of all sequences. This type typically appears in programs via the combination of ocurrence typing ang `sequence?`.

Example:

```
> (lambda: ([x : Any]) (if (sequence? x) x (error "not a
sequence!")))
- : (-> Any SequenceTop)
#<procedure>
```

Added in version 1.10 of package `typed-racket-lib`.

```
(Custodian-Boxof t)
```

Returns the type of custodian box of `t`.

```
(Thread-Cellof t)
```

Returns the type of thread cell of `t`.

Thread-CellTop

Is the type of a thread cell with unknown element type and is the supertype of all thread cell types. This type typically appears in programs via the combination of occurrence typing and `thread-cell?`.

Example:

```
> (lambda: ([x : Any]) (if (thread-cell? x) x (error "not a thread
cell!")))
- : (-> Any Thread-CellTop)
#<procedure>
```

(Weak-Boxof *t*)

Returns the type for a weak box whose value is of type *t*.

Examples:

```
> (make-weak-box 5)
- : (Weak-Boxof Integer)
#<weak-box>
> (weak-box-value (make-weak-box 5))
- : (U False Integer)
5
```

Weak-BoxTop

Is the type of a weak box with an unknown element type and is the supertype of all weak box types. This type typically appears in programs via the combination of occurrence typing and `weak-box?`.

Example:

```
> (lambda: ([x : Any]) (if (weak-box? x) x (error "not a box!")))
- : (-> Any Weak-BoxTop)
#<procedure>
```

(Ephemeronof *t*)

Returns the type of an ephemeron whose value is of type *t*.

(Evtof *t*)

A synchronizable event whose synchronization result is of type `t`.

Examples:

```
> always-evt
- : (Rec x (Evtof x))
#<always-evt>
> (system-idle-evt)
- : (Evtof Void)
#<system-idle-evt>
> (ann (thread (λ () (displayln "hello world"))) (Evtof Thread))
- : (Evtof Thread)
hello world
#<thread>
```

1.4 Syntax Objects

The following type constructors and types respectively create and represent syntax objects and their content.

```
(Syntaxof t)
```

Returns the type of 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, byte 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)
```

Returns the recursive union of `t` with symbols, keywords, strings, byte 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 type constructors and type respectively create and represent prompt tags and keys for continuation marks for use with delimited continuation functions and continuation mark functions.

`(Prompt-Tagof s t)`

Returns the type of 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 Any Any)
#<continuation-prompt-tag:prompt-tag>
```

`Prompt-TagTop`

is the type of a prompt tag with unknown body and handler types and is the supertype of all prompt tag types. This type typically appears in programs via the combination of occurrence typing and `continuation-prompt-tag?`.

Example:

```
> (lambda: ([x : Any]) (if (continuation-prompt-
tag? x) x (error "not a prompt tag!")))
- : (-> Any Prompt-TagTop)
#<procedure>
```

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

Returns the type of 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 Any)
#<continuation-mark-key>
```

```
Continuation-Mark-KeyTop
```

Is the type of a continuation mark key with unknown element type and is the supertype of all continuation mark key types. This type typically appears in programs via the combination of occurrence typing and `continuation-mark-key?`.

Example:

```
> (lambda: ([x : Any]) (if (continuation-mark-
key? x) x (error "not a mark key!")))
- : (-> Any Continuation-Mark-KeyTop)
#<procedure>
```

1.6 Other Type Constructors

```
(-> dom ... rng opt-proposition)
(-> dom ... rest * rng)
(-> dom ... rest ooo bound rng)
(dom ... -> rng opt-proposition)
(dom ... rest * -> rng)
(dom ... rest ooo bound -> rng)
```

```
            ooo = ...

            dom = type
                | mandatory-kw
                | opt-kw

            rng = type
                | (Some (a ...) type : #:+ proposition)
                | (Values type ...)

   mandatory-kw = keyword type

         opt-kw = [keyword type]

opt-proposition =
                | : type
                | : pos-proposition
                    neg-proposition
                  object

pos-proposition =
                | #:+ proposition ...

neg-proposition =
                | #:- proposition ...

         object =
                | #:object index

    proposition = Top
                | Bot
                | type
                | (! type)
                | (type @ path-elem ... index)
                | (! type @ path-elem ... index)
                | (and proposition ...)
                | (or proposition ...)
                | (implies proposition ...)

      path-elem = car
                | cdr

          index = positive-integer
                | (positive-integer positive-integer)
                | identifier
```

The type of functions from the (possibly-empty) sequence *dom* ... to the *rng* type.

Examples:

```
> (λ ([x : Number]) x)
- : (-> Number Number)
#<procedure>
> (λ () 'hello)
- : (-> 'hello)
#<procedure>
```

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*. The bound refers to the type variable that is in scope within the rest argument type.

Examples:

```
> (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>
```

In the third form, the ... introduced by *ooo* is literal, and *bound* must be an identifier denoting a type variable.

The *dom*s can include both mandatory and optional keyword arguments. Mandatory keyword arguments are a pair of keyword and type, while optional arguments are surrounded by a pair of parentheses.

Examples:

```
> (: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
```

When *opt-proposition* is provided, it specifies the *proposition* for the function type (for an introduction to propositions in Typed Racket, see §5.2 "Propositions and Predicates"). For almost all use cases, only the simplest form of propositions, with a single type after a `:`, are necessary:

Example:

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

The proposition specifies that when `(string? x)` evaluates to a true value for a conditional branch, the variable `x` in that branch can be assumed to have type `String`. Likewise, if the expression evaluates to `#f` in a branch, the variable *does not* have type `String`.

In some cases, asymmetric type information is useful in the propositions. For example, the `filter` function's first argument is specified with only a positive proposition:

Example:

```
> filter
- : (All (a b)
      (case->
       (-> (-> a Any : #:+ b) (Listof a) (Listof b))
       (-> (-> a Any) (Listof a) (Listof a))))
#<procedure:filter>
```

The use of `#:+` indicates that when the function applied to a variable evaluates to a true value, the given type can be assumed for the variable. However, the type-checker gains no information in branches in which the result is `#f`.

Conversely, `#:-` specifies that a function provides information for the false branch of a conditional.

The other proposition cases are rarely needed, but the grammar documents them for completeness. They correspond to logical operations on the propositions.

The type of functions can also be specified with an *infix* `->` which comes immediately before the *rng* type. The fourth through sixth forms match the first three cases, but with the infix style of arrow.

Examples:

```
> (: add2 (Number -> Number))
> (define (add2 n) (+ n 2))
```

Currently, because explicit packing operations for existential types are not supported, existential type results are only used to annotate accessors for `Struct-Property`

`(Some (a ...) type : #:+ proposition)` for `rng` specifies an *existential type result*, where the type variables `a ...` may appear in `type` and `opt-proposition`. Unpacking the existential type result is done automatically while checking application of the function.

Changed in version 1.12 of package `typed-racket-lib`: Added existential type results

```
(->* (mandatory-dom ...) optional-doms rest rng)

mandatory-dom = type
              | keyword type

optional-doms =
              | (optional-dom ...)

 optional-dom = type
              | keyword type

         rest =
              | #:rest type
              | #:rest-star (type ...)
```

Constructs the type of functions with optional or rest arguments. The first list of `mandatory-dom`s correspond to mandatory argument types. The list `optional-doms`, if provided, specifies the optional argument types.

Examples:

```
> (: append-bar (->* (String) (Positive-Integer) String))
> (define (append-bar str [how-many 1])
    (apply string-append str (make-list how-many "bar")))
```

If provided, the `#:rest type` specifies the type of elements in the rest argument list.

Examples:

```
> (: +all (->* (Integer) #:rest Integer (Listof Integer)))
> (define (+all inc . rst)
    (map (λ ([x : Integer]) (+ x inc)) rst))
> (+all 20 1 2 3)
- : (Listof Integer)
'(21 22 23)
```

A `#:rest-star (type ...)` specifies the rest list is a sequence of types which occurs 0 or more times (i.e. the Kleene closure of the sequence).

Examples:

```
> (: print-name+ages (->* () #:rest-star (String Natural) Void))
> (define (print-name+ages . names+ages)
    (let loop ([names+ages : (Rec x (U Null (List* String Natural x))) names+ages])
      (when (pair? names+ages)
        (printf "~a is ~a years old!\n"
                (first names+ages)
                (second names+ages))
        (loop (cddr names+ages))))
    (printf "done printing ~a ages" (/ (length names+ages) 2)))
> (print-name+ages)
done printing 0 ages
> (print-name+ages "Charlotte" 8 "Harrison" 5 "Sydney" 3)
Charlotte is 8 years old!
Harrison is 5 years old!
Sydney is 3 years old!
done printing 3 ages
```

Both the mandatory and optional argument lists may contain keywords paired with types.

Examples:

```
> (: kw-f (->* (#:x Integer) (#:y Integer) Integer))
> (define (kw-f #:x x #:y [y 0]) (+ x y))
```

The syntax for this type constructor matches the syntax of the `->*` contract combinator, but with types instead of contracts.

```
Top
Bot
```

These are propositions that can be used with `->`. `Top` is the propositions with no information. `Bot` is the propositions which means the result cannot happen.

```
Procedure
```

is the supertype of all function types. The `Procedure` type corresponds to values that satisfy the `procedure?` predicate. Because this type encodes *only* the fact that the value is a procedure, and *not* its argument types or even arity, the type-checker cannot allow values of this type to be applied.

For the types of functions with known arity and argument types, see the `->` type constructor.

Examples:

```
> (: my-list Procedure)
```

```
> (define my-list list)
> (my-list "zwiebelkuchen" "socca")
eval:91:0: Type Checker: cannot apply a function with
unknown arity;
 function `my-list' has type Procedure which cannot be
applied
   in: "socca"
```

`(U t ...)`

is the union of the types `t ...`.

Example:

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

`(∩ t ...)`

is the intersection of the types `t ...`.

Example:

```
> ((λ #:forall (A) ([x : (∩ Symbol A)]) x) 'foo)
- : 'foo
'foo
```

`(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 non-dependent function types (i.e. no preconditions or dependencies between arguments are currently allowed).

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 (a ...) t)
(All (a ... a ooo) t)
```

is a parameterization of type `t`, with type variables `a ...`. If `t` is a function type constructed with infix `->`, 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 [more precisely: Nonnegative-Integer]
3
```

```
(Some (a ...) t)
```

See existential type results.

Added in version 1.10 of package `typed-racket-lib`.

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

Returns 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) [more precisely: (Values One
Positive-Byte Positive-Byte)]
1
2
3
```

Note that a type variable cannot be instantiated with a `(Values ...)` type. For example, the type `(All (A) (-> A))` describes a thunk that returns exactly one value.

```
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.

```
(Struct-Type st)
```

is a type for the structure type descriptor value for the structure type `st`. Values of this type are used with reflective operations such as `struct-type-info`.

Examples:

```
> struct:arity-at-least
- : (StructType arity-at-least)
#<struct-type:arity-at-least>
> (struct-type-info struct:arity-at-least)
- : (values
     Symbol
     Integer
     Integer
     (-> arity-at-least Nonnegative-Integer Any)
     (-> arity-at-least Nonnegative-Integer Nothing Void)
     (Listof Nonnegative-Integer)
     (U False Struct-TypeTop)
```

```
     Boolean)
[more precisely: (values
                  Symbol
                  Nonnegative-Integer
                  Nonnegative-Integer
                  (-> arity-at-least Nonnegative-Integer Any)
                  (-> arity-at-least Nonnegative-Integer Nothing
Void)
                  (Listof Nonnegative-Integer)
                  (U False Struct-TypeTop)
                  Boolean)]
'arity-at-least
1
0
#<procedure:arity-at-least-ref>
#<procedure:arity-at-least-set!>
'(0)
#f
#f
```

```
Struct-TypeTop
```

is the supertype of all types for structure type descriptor values. The corresponding structure type is unknown for values of this top type.

Example:

```
> (struct-info (arity-at-least 0))
- : (values (U False Struct-TypeTop) Boolean)
#<struct-type:arity-at-least>
#f
```

```
(Prefab key type ...)
```

Describes a prefab structure with the given (implicitly quoted) *prefab key* `key` and specified field types.

Prefabs are more-or-less tagged polymorphic tuples which can be directly serialized and whose fields can be accessed by anyone. Subtyping is covariant for immutable fields and invariant for mutable fields.

When a prefab struct is defined with `struct` the struct name is bound at the type-level to the `Prefab` type with the corresponding key and field types and the constructor expects types corresponding to those declared for each field. The defined predicate, however, only tests whether a value is a prefab structure with the same key and number of fields, but does not inspect the fields' values.

Examples:

```
> (struct person ([name : String]) #:prefab)
> person
- : (-> String person)
#<procedure:person>
> person?
- : (-> Any Boolean : (Prefab person Any))
#<procedure:person?>
> person-name
- : (All (x) (case-> (-> (Prefab person x) x) (-> (Prefab person
Any) Any)))
#<procedure:person-name>
> (person "Jim")
- : (Prefab person String)
'#s(person "Jim")
> (ann '#s(person "Dwight") person)
- : (Prefab person String)
'#s(person "Dwight")
> (ann '#s(person "Pam") (Prefab person String))
- : person
'#s(person "Pam")
> (ann '#s(person "Michael") (Prefab person Any))
- : (Prefab person Any)
'#s(person "Michael")
> (person 'Toby)
eval:112:0: Type Checker: type mismatch
  expected: String
  given: 'Toby
  in: Toby
> (ann #s(person Toby) (Prefab person String))
eval:113:0: Type Checker: type mismatch
  expected: person
  given: (Prefab person 'Toby)
  in: String
> (ann '#s(person Toby) (Prefab person Symbol))
- : (Prefab person Symbol)
'#s(person Toby)
> (person? '#s(person "Michael"))
- : True
#t
> (person? '#s(person Toby))
- : True
#t
> (struct employee person ([schrute-bucks : Natural]) #:prefab)
> (employee "Oscar" 10000)
```

```
- : (Prefab (employee person 1) String Nonnegative-Integer)
'#s((employee person 1) "Oscar" 10000)
> (ann '#s((employee person 1)             "Oscar" 10000) employee)
- : (Prefab (employee person 1) String Nonnegative-Integer)
'#s((employee person 1) "Oscar" 10000)
> (ann '#s((employee person 1)             "Oscar" 10000)
       (Prefab (employee person 1) String Natural))
- : employee
'#s((employee person 1) "Oscar" 10000)
> (person? '#s((employee person 1)             "Oscar" 10000))
- : True
#t
> (employee? '#s((employee person 1)             "Oscar" 10000))
- : True
#t
> (employee 'Toby -1)
eval:123:0: Type Checker: type mismatch
  expected: String
  given: 'Toby
  in: -1
> (ann '#s((employee person 1)             Toby -1)
       (Prefab (employee person 1) Symbol Integer))
- : (Prefab (employee person 1) Symbol Integer)
'#s((employee person 1) Toby -1)
> (person? '#s((employee person 1)             Toby -1))
- : True
#t
> (employee? '#s((employee person 1)             Toby -1))
- : True
#t
```

```
(PrefabTop key field-count)
```

Describes all prefab types with the (implicitly quoted) prefab-key *key* and *field-count* many fields.

For immutable prefabs this is equivalent to `(Prefab key Any ...)` with *field-count* many occurrences of `Any`. For mutable prefabs, this describes a prefab that can be read from but not written to (since we do not know at what type other code may have the fields typed at).

Examples:

```
> (struct point ([x : Number] [y : Number])
    #:prefab
    #:mutable)
```

```
> point
- : (-> Number Number point)
#<procedure:point>
> point-x
- : (All (a b)
      (case->
       (-> (Prefab (point #(0 1)) a b) a)
       (-> (PrefabTop (point #(0 1)) 2) Any)))
#<procedure:point-x>
> point-y
- : (All (a b)
      (case->
       (-> (Prefab (point #(0 1)) a b) b)
       (-> (PrefabTop (point #(0 1)) 2) Any)))
#<procedure:point-y>
> point?
- : (-> Any Boolean : (PrefabTop (point #(0 1)) 2))
#<procedure:point?>
> (define (maybe-read-x p)
    (if (point? p)
        (ann (point-x p) Any)
        'not-a-point))
> (define (read-some-x-num p)
  (if (point? p)
      (ann (point-x p) Number)
      -1))
eval:133:0: Type Checker: Polymorphic function `point-x'
could not be applied to arguments:
Types: (PrefabTop (point #(0 1)) 2)   -> Any
Arguments: (PrefabTop (point #(0 1)) 2)
Expected result: Number

  in: -1
```

Added in version 1.7 of package `typed-racket-lib`.

```
(Struct-Property ty)
```

Describes a property that can be attached to a structure type. The property value must match the type *ty*.

Example:

```
> (:print-type prop:input-port)
(Struct-Property (U Exact-Nonnegative-Integer Input-Port))
```

Added in version 1.10 of package `typed-racket-lib`.

`Self`

This type can only appear in a `Struct-Property` type. A struct property value is attached to an instance of a structure type; the `Self` type refers to this instance.

Example:

```
> (:print-type prop:custom-write)
(Struct-Property (-> Self Output-Port (U Boolean One Zero) AnyVal-
ues))
```

Added in version 1.10 of package `typed-racket-lib`.

`Imp`

This type can only appear in a `Struct-Property` type. An `Imp` value may be a structure subtype of the `Self` value, or another instance created by the same struct constructor.

Example:

```
> (:print-type prop:equal+hash)
(Struct-Property
 (List
  (-> Self Imp (-> Any Any Boolean) Any)
  (-> Self (-> Any Integer) Integer)
  (-> Self (-> Any Integer) Integer)))
```

Added in version 1.10 of package `typed-racket-lib`.

`(Has-Struct-Property` *prop*`)`

This type describes an instance of a structure type associcated with a `Struct-Property` named *prop*.

`∪`

An alias for `U`.

`Union`

An alias for `U`.

`Intersection`

An alias for ∩.

`→`

An alias for `->`.

`→*`

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 the `#:opaque` clause of `require/typed`.

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 *var* are names, *type* is a type. *e* is an expression and *body* is a block.

```
(let maybe-tvars (binding ...) maybe-ret . body)
(let loop maybe-ret (binding ...) . body)

    binding = [var e]
            | [var : type e]

maybe-tvars =
            | #:forall (tvar ...)
            | #:∀ (tvar ...)

  maybe-ret =
            | : type0
```

Local bindings, like `let`, each with associated types.

In the first form, *maybe-ret* can only appear with *maybe-tvars*, so if you only want to specify the return type, you should set *maybe-tvars* to `#:forall ()`.

Examples:

```
> (let ([x : Zero 0]) x)
- : Integer [more precisely: Zero]
0
> (let #:forall () ([x : Zero 0]) : Natural x)
- : Integer [more precisely: Zero]
0
> (let ([x : Zero 0]) : Natural x)
eval:4:0: :: bad syntax
  in: :
```

If polymorphic type variables are provided, they are bound in the type expressions for variable bindings.

Example:

```
> (let #:forall (A) ([x : A 0]) x)
- : Integer [more precisely: Zero]
0
```

In the second form, *type0* 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)
```

```
(letrec (binding ...) . body)
(let* (binding ...) . body)
(let-values ([(var+type ...) e] ...) . body)
(letrec-values ([(var+type ...) e] ...) . body)
(let*-values ([(var+type ...) 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`. As with `let`, the type annotation is optional.

2.2 Anonymous Functions

```
(lambda maybe-tvars formals maybe-ret . body)

    formals = (formal ...)
            | (formal ... . rst)

     formal = var
            | [var default-expr]
            | [var : type]
            | [var : type default-expr]
            | keyword var
            | keyword [var : type]
            | keyword [var : type default-expr]

        rst = var
            | [var : type *]
            | [var : type ooo bound]

maybe-tvars =
            | #:forall (tvar ...)
            | #:∀ (tvar ...)
            | #:forall (tvar ... ooo)
            | #:∀ (tvar ... ooo)

  maybe-ret =
            | : type
```

Constructs an anonymous function like the `lambda` form from `racket/base`, but allows type annotations on the formal arguments. If a type annotation is left out, the formal will have the type `Any`.

Examples:

```
> (lambda ([x : String]) (string-append x "bar"))
- : (-> String String)
#<procedure>
```

```
> (lambda (x [y : Integer]) (add1 y))
- : (-> Any Integer Integer)
#<procedure>
> (lambda (x) x)
- : (-> Any Any)
#<procedure>
```

Type annotations may also be specified for keyword and optional arguments:

Examples:

```
> (lambda ([x : String "foo"]) (string-append x "bar"))
- : (->* () (String) (String : (Top | Bot)))
#<procedure:eval:15:0>
> (lambda (#:x [x : String]) (string-append x "bar"))
- : (-> #:x String String)
#<procedure:eval:16:0>
> (lambda (x #:y [y : Integer 0]) (add1 y))
- : (-> Any [#:y Integer] Integer)
#<procedure:eval:17:0>
> (lambda ([x 'default]) x)
- : (->* () (Any) Any)
#<procedure:eval:18:0>
```

The `lambda` expression may also specify polymorphic type variables that are bound for the type expressions in the formals.

Examples:

```
> (lambda #:forall (A) ([x : A]) x)
- : (All (A) (-> A A))
#<procedure>
> (lambda #:∀ (A) ([x : A]) x)
- : (All (A) (-> A A))
#<procedure>
```

In addition, a type may optionally be specified for the rest argument with either a uniform type or using a polymorphic type. In the former case, the rest argument is given the type `(Listof` *type*`)` where *type* is the provided type annotation.

Examples:

```
> (lambda (x . rst) rst)
- : (-> Any Any * (Listof Any))
#<procedure>
```

```
> (lambda (x    rst : Integer *)  rst)
- : (-> Any Integer * (Listof Integer))
#<procedure>
> (lambda #:forall (A ...) (x    rst : A ... A)  rst)
- : (All (A ...) (-> Any A ... A (List A ... A)))
#<procedure>
```

λ

An alias for `lambda`.

```
(case-lambda maybe-tvars [formals body] ...)
```

A function of multiple arities. The *formals* are identical to those accepted by the `lambda` form except that keyword and optional arguments are not allowed.

Polymorphic type variables, if provided, are bound in the type expressions in the formals.

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)]))
```

To see how to declare a type for `add-map`, see the `case->` type constructor.

case-λ

An alias for `case-lambda`.

2.3 Loops

```
(for void-ann-maybe (for-clause ...) void-ann-maybe expr ...+)
```

```
void-ann-maybe =
               | : Void

type-ann-maybe =
               | : u

    for-clause = [id : t seq-expr]
               | [(binding ...) seq-expr]
               | [id seq-expr]
               | #:when guard
               | #:unless guard
               | #:do [do-body ...]
               | break-clause
               | #:splice (splicing-id . form)

       binding = id
               | [id : t]

  break-clause = #:break guard
               | #:final guard
```

Like `for` from `racket/base`, but each *id* has the associated type *t*. The latter *ann-maybe* will be used first, and then the previous one. Since the return type is always *Void*, annotating the return type of a `for` form is optional. Type annotations in clauses are optional for all `for` variants.

Examples:

```
> (for ([i '()]) i)
> (for : Void ([i '()]) i)
> (for ([i '()]) : Void i)
> (for : Void ([i '()]) : Void i)
> (for ([i '()]) : Any i)
eval:29:0: :: bad syntax
  in: :
> (for/or : False ([i '()]) : False #f)
- : False
#f
> (for/or : Boolean ([i '()]) : False #f)
- : Boolean
#f
> (for/or : False ([i '()]) : Boolean #f)
eval:32:0: Type Checker: type mismatch
  expected: False
  given: Boolean
  in: #f
```

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

Like the above, except they are not yet supported by the typechecker.

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

maybe-result =
             | #:result result-expr
```

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

Changed in version 1.11 of package `typed-racket-lib`: Added the `#:result` form.

Changed in version 1.12 of package `typed-racket-lib`: Added `for/foldr`.

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

maybe-result =
             | #:result result-expr
```

These behave like their non-annotated counterparts.

Changed in version 1.11 of package `typed-racket-lib`: Added the `#:result` form.

Changed in version 1.12 of package `typed-racket-lib`: Added `for*/foldr`.

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

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

Like `do` from `racket/base`, 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 maybe-tvars v maybe-ann e)
(define maybe-tvars header maybe-ann . body)
```

```
    header = (function-name . formals)
           | (header . formals)

   formals = (formal ...)
           | (formal ... . rst)

    formal = var
           | [var default-expr]
           | [var : type]
           | [var : type default-expr]
           | keyword var
           | keyword [var : type]
           | keyword [var : type default-expr]

       rst = var
           | [var : type *]
           | [var : type ooo bound]

maybe-tvars =
           | #:forall (tvar ...)
           | #:∀ (tvar ...)
           | #:forall (tvar ... ooo)
           | #:∀ (tvar ... ooo)

 maybe-ann =
           | : type
```

Like `define` from `racket/base`, but allows optional type annotations for the variables.

The first form defines a variable `v` to the result of evaluating the expression `e`. The variable may have an optional type annotation.

Examples:

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

If polymorphic type variables are provided, then they are bound for use in the type annotation.

Example:

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

The second form allows the definition of functions with optional type annotations on any variables. If a return type annotation is provided, it is used to check the result of the function.

Like `lambda`, optional and keyword arguments are supported.

Examples:

```
> (define (add [first : Integer]
               [second  : Integer]) : Integer
    (+ first second))
> (define #:forall (A)
          (poly-app [func : (A A -> A)]
                    [first : A]
                    [second  : A]) : A
    (func first second))
```

The function definition form also allows curried function arguments with corresponding type annotations.

Examples:

```
> (define ((addx [x : Number]) [y : Number]) (+ x y))
> (define add2 (addx 2))
> (add2 5)
- : Number
7
```

Note that unlike `define` from `racket/base`, `define` does not bind functions with keyword arguments to static information about those functions.

2.5 Structure Definitions

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

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

      name-spec = name-id
                | name-id parent

        options = #:transparent
                | #:mutable
                | #:prefab
                | #:constructor-name constructor-id
                | #:extra-constructor-name constructor-id
                | #:property property-id property-expr
                | #:type-name type-id
```

Defines a structure with the name *name-id*, where the fields *f* have types *t*, similar to the behavior of `struct` from `racket/base`.

Examples:

```
> (struct camelia-sinensis ([age : Integer]))
> (struct camelia-sinensis-assamica camelia-sinensis ())
```

If *type-id* is not specified, *name-id* will be used for the name of the type associated with instances of the declared structure. Otherwise, *type-id* will be used for the type name, and using *name-id* in this case will cause a type error.

Examples:

```
> (struct apple () #:type-name BigApple)
> (ann (apple) BigApple)
- : BigApple
#<apple>
> (ann (apple) apple)
eval:45:0: Type Checker: parse error in type;
 type name `apple' is unbound
  in: apple
```

type-id can be also used as an alias to *name-id*, i.e. it will be a transformer binding that encapsulates the same structure information as *name-id* does.

Examples:

```
> (struct avocado ([amount : Integer]) #:type-name Avocado)
> (struct hass-avocado Avocado ())
> (struct-copy Avocado (avocado 0) [amount 42])
- : Avocado
#<avocado>
```

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.

Examples:

```
> (struct (X Y) 2-tuple ([first : X] [second : Y]))
> (struct (X Y Z) 3-tuple 2-tuple ([third :  Z]))
```

Options provided have the same meaning as for the `struct` form from `racket/base` (with the exception of `#:type-name`, as described above).

A prefab structure type declaration will bind the given *name-id* or *type-id* to a `Prefab` type. Unlike the `struct` form from `racket/base`, a non-prefab structure type cannot extend a prefab structure type.

Examples:

```
> (struct a-prefab ([x : String]) #:prefab)
> (:type a-prefab)
(Prefab a-prefab String)
> (struct not-allowed a-prefab ())
eval:53:0: Type Checker: Error in macro expansion -- parent
type not a valid structure name: a-prefab
  in: ()
```

Changed in version 1.4 of package `typed-racket-lib`: Added the `#:type-name` option.

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

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

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

        options = #:transparent
                | #:mutable
                | #:type-name type-id
```

Legacy version of `struct`, corresponding to `define-struct` from `racket/base`.

Changed in version 1.4 of package `typed-racket-lib`: Added the `#:type-name` option.

2.6 Names for Types

```
(define-type name t maybe-omit-def)
(define-type (name v ...) t maybe-omit-def)

maybe-omit-def = #:omit-define-syntaxes
               |
```

The first form defines *name* as type, with the same meaning as *t*. The second form defines *name* to be a type constructor, whose parameters are *v* ... and body is *t*. If no parameters

are declared, the defined type constructor is equivalent to `(define-type name t maybe-omit-def)`. Type names may refer to other types defined in the same or enclosing scopes.

Examples:

```
> (define-type IntStr (U Integer String))
> (define-type (ListofPairs A) (Listof (Pair A A)))
```

If `#:omit-define-syntaxes` is specified, no definition of *name* is created. In this case, some other definition of *name* is necessary.

If the body of the type definition refers to itself, then the type definition is recursive. Recursion may also occur mutually, if a type refers to a chain of other types that eventually refers back to itself.

Examples:

```
> (define-type BT (U Number (Pair BT BT)))
> (let ()
    (define-type (Even A) (U Null (Pairof A (Odd A))))
    (define-type (Odd A) (Pairof A (Even A)))
    (: even-lst (Even Integer))
    (define even-lst '(1 2))
    even-lst)
- : (Even Integer)
'(1 2)
```

However, the recursive reference is only allowed when it is passed to a productive type constructor:

Examples:

```
> (define-type Foo Foo)
eval:58:0: Type Checker: Error in macro expansion -- parse
error in type;
 not in a productive position
   variable: Foo
   in: Foo
> (define-type Bar (U Bar False))
eval:59:0: Type Checker: Error in macro expansion -- parse
error in type;
 not in a productive position
   variable: Bar
   in: False
> (define-type Bar (U (Listof Bar) False))
```

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)
(: 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)
```

The second form allows type annotations to elide one level of parentheses for function types.

Examples:

```
> (: var3 : -> Integer)
> (: var4 : String -> Integer)
```

```
(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`.

If a dispatch macro on `#\{` already exists in the current readtable, this syntax will be disabled.

```
(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)`.

If a dispatch macro on `#\{` already exists in the current readtable, this syntax will be disabled.

```
(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)
(cast for #f): broke its own contract
  promised: string?
  produced: 3
  in: string?
  contract from: cast
  blaming: cast
   (assuming the contract is correct)
  at: eval:66:0
> (cast (lambda ([x : Any]) x) (String -> String))
- : (-> String String)
#<procedure:val>
> ((cast (lambda ([x : Any]) x) (String -> String)) "hello")
- : String
"hello"
```

The value is actually protected with two contracts. The second contract checks the new type, but the first contract is put there to enforce the old type, to protect higher-order uses of the value.

Examples:

```
> ((cast (lambda ([s : String]) s) (Any -> Any)) "hello")
- : Any
"hello"
> ((cast (lambda ([s : String]) s) (Any -> Any)) 5)
(cast for val): contract violation
  expected: string?
  given: 5
  in: the 1st argument of
      (-> string? any)
  contract from: typed-world
  blaming: cast
   (assuming the contract is correct)
  at: eval:70:0
```

`cast` will wrap the value `e` in a contract which will affect the runtime performance of reading and updating the value. This is needed when `e` is a complex data type, such as a hash table. However, when the type of the value can be checked using a simple predicate, consider using `assert` instead.

```
(inst e t ...)
(inst e t ... t ooo bound)
```

Instantiate the type of `e` with types `t ...` or with the poly-dotted types `t ... t ooo bound`. `e` must have a polymorphic type that can be applied to the supplied number of type variables. For non-poly-dotted functions, however, fewer arguments can be provided and the omitted types default to `Any`. `inst` is legal only in expression contexts.

Examples:

```
> (foldl (inst cons Integer Integer) null (list 1 2 3 4))
- : (Listof Integer)
'(4 3 2 1)
> (: my-cons (All (A B) (-> A B (Pairof A B))))
> (define my-cons cons)
> (: foldl-list : (All (α) (Listof α) -> (Listof α)))
> (define (foldl-list lst)
    (foldl (inst my-cons α (Listof α)) null lst))
> (foldl-list (list "1" "2" "3" "4"))
- : (Listof String)
'("4" "3" "2" "1")
> (: foldr-list : (All (α) (Listof α) -> Any))
> (define (foldr-list lst)
    (foldr (inst my-cons α) null lst))
> (foldr-list (list "1" "2" "3" "4"))
- : Any
'("1" "2" "3" "4")
```

```
> (: my-values : (All (A B ...) (A B ... -> (values A B ... B))))
> (define (my-values arg . args)
    (apply (inst values A B ... B) arg args))
```

```
(row-inst e row)
```

Instantiate the row-polymorphic type of *e* with *row*. This is legal only in expression contexts.

Examples:

```
> (: id (All (r #:row)
           (-> (Class #:row-var r) (Class #:row-var r))))
> (define (id cls) cls)
> ((row-inst id (Row (field [x Integer])))
   (class object% (super-new) (field [x : Integer 0])))
- : (Class (field (x Integer)))
#<class:eval:84:0>
```

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

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

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

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

2.9 Require

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

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

```
    rt-clause = [maybe-renamed t]
              | [#:struct maybe-tvars name-id ([f : t] ...)
                    struct-option ...]
              | [#:struct maybe-tvars (name-id parent) ([f : t] ...)
                    struct-option ...]
              | [#:opaque t pred]
              | [#:signature name ([id : t] ...)]

maybe-renamed = id
              | (orig-id new-id)

  maybe-tvars =
              | (type-variable ...)

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

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

The first case requires *maybe-renamed*, giving it type `t`.

The second and third cases require the struct with name `name-id` and creates a new type with the name `type-id`, or `name-id` if no `type-id` is provided, 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 *opaque type* `t` using the function `pred` as a predicate. (Module `m` must provide `pred` and `pred` must have type `(Any -> Boolean)`.) The type `t` is defined as precisely those values that `pred` returns `#t` for. Opaque types must be required lexically before they are used.

Examples:

```
> (require/typed racket/base
                 [#:opaque Evt evt?]
                 [alarm-evt (Real -> Evt)]
                 [sync (Evt -> Any)])
> evt?
- : (-> Any Boolean : Evt)
#<procedure:evt?>
> (sync (alarm-evt (+ 100 (current-inexact-milliseconds))))
- : Any
#<alarm-evt>
```

The `#:signature` keyword registers the required signature in the signature environment. For more information on the use of signatures in Typed Racket see the documentation for `typed/racket/unit`.

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->`.

Changed in version 1.4 of package `typed-racket-lib`: Added the `#:type-name` option.

Changed in version 1.6: Added syntax for struct type variables, only works in unsafe requires.

Changed in version 1.12: Added default type `Any` for omitted `inst` args.

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

Similar to `require/typed`, but also provides the imported identifiers. Uses outside of a module top-level raise an error.

Examples:

```
> (module evts typed/racket
    (require/typed/provide racket/base
                           [#:opaque Evt evt?]
                           [alarm-evt (Real -> Evt)]
                           [sync (Evt -> Any)]))
> (require 'evts)
> (sync (alarm-evt (+ 100 (current-inexact-milliseconds))))
- : Any
#<alarm-evt>
```

2.10 Other Forms

```
with-handlers
```

Identical to `with-handlers` from `racket/base` but provides additional annotations to assist the typechecker.

```
with-handlers*
```

Identical to `with-handlers*` from `racket/base` but provides additional annotations to assist the typechecker.

Added in version 1.12 of package `typed-racket-lib`.

```
(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 own 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
   (assuming the contract is correct)
```

(#%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.

2.11 Special Structure Type Properties

prop:procedure : struct-type-property?

Unlike many other structure type properties, `prop:procedure` does not have predefined types for its property values. When a structure is assocatied with `prop:procedure`, its constructors' return type is an intersection type of the structure type and a function type specified by the property value.

Examples:

```
> (struct animal ([a : Number] [b : (-> Number Number)])
    #:property prop:procedure
    (struct-field-index b))
> (animal 2 add1)
- : (∩ (-> Number Number) animal)
#<procedure:add1>
> (struct plant ([a : Number])
    #:property prop:procedure
    (lambda ([me : plant] [a1 : String]) : Number
      (+ (plant-a me) (string-length a1))))
> (plant 31)
- : (∩ (-> String Number) plant)
#<procedure:plant>
```

In other words, a variable that refers to a function is not allowed

Unlike in Racket, only one of the following types of expressions are allowed in Typed Racket: a nonnegative literal, `(struct-index-field field-name)`, or a lambda expression. Note that in the last case, if the type annotation on the codomain is not supplied, the type checker will use `Any` as the return type.

Similar to other structure type properties, when a structure's base structure specifies a value for `prop:procedure`, the structure inherits that value if it does not specify its own.

Examples:

```
> (struct cat animal ([c : Number]))
> (cat 2 add1 42)
- : (∩ (-> Number Number) cat)
#<procedure:add1>
> (struct a-cat cat ())
> (a-cat 2 add1 42)
- : (∩ (-> Number Number) a-cat)
#<procedure:add1>
```

Function types for procedural structures do not enforce subtyping relations. A substructure can specify a different field index or a procedure that has a arity and/or types different from its base structures for `prop:procedure`.

Examples:

```
> (struct b-cat cat ([d : (-> Number String)])
    #:property prop:procedure (struct-field-index d))
> (b-cat 2 add1 42 number->string)
- : (∩ (-> Number String) b-cat)
#<procedure:number->string>
```

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"))
```

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)
```

The following libraries are included with Typed Racket in the `typed` collection:

Typed for typed/file/gif

`(require typed/file/gif)` package: `typed-racket-more`

`GIF-Stream`

Describe a GIF stream, as produced by `gif-start` and accepted by the other functions from `file/gif`.

`GIF-Colormap`

Type alias for a list of three-element (R,G,B) vectors representing an image.

Typed for typed/file/md5

```
(require typed/file/md5)
```
package: typed-racket-lib

Typed for typed/file/sha1

```
(require typed/file/sha1)
```
package: typed-racket-lib

Typed for typed/file/tar

```
(require typed/file/tar)
```
package: typed-racket-lib

Typed for typed/framework

```
(require typed/framework)
```
package: typed-racket-more

Typed for typed/json

```
(require typed/json)
```
package: typed-racket-more

Unlike the untyped `json` library, `typed/json` always uses `'null` to represent the JSON "null" value. The functions exported by `typed/json` do not accept a `#:null` argument, and they are not sensitive to the current value of the `json-null` parameter. The `json-null` binding itself is not exported by `typed/json`.

```
JSExpr
```

Describes a jsexpr, where `'null` is always used to represent the JSON "null" value.

Typed for typed/mred/mred

```
(require typed/mred/mred)
```
package: typed-racket-more

Typed for typed/net/base64

```
(require typed/net/base64)
```
package: typed-racket-more

Typed for typed/net/cgi

```
(require typed/net/cgi)
```
package: typed-racket-more

Typed for typed/net/cookies

```
(require typed/net/cookies)
```
package: typed-racket-more

Typed for typed/net/cookies/common

```
(require typed/net/cookies/common)
```
package: typed-racket-more

Added in version 1.10 of package typed-racket-lib.

Typed for typed/net/cookies/server

```
(require typed/net/cookies/server)
```
package: typed-racket-more

Cookie

Describes a server-side RFC 6265 HTTP cookie, as implemented by net/cookies/server.

Added in version 1.10 of package typed-racket-lib.

Typed for typed/net/cookie

```
(require typed/net/cookie)
```
package: typed-racket-more

> **NOTE:** This library is deprecated; use typed/net/cookies, instead. This library is deprecated for the same reasons that net/cookie is deprecated.

Cookie

Describes an HTTP cookie as implemented by net/cookie, which is deprecated in favor of net/cookies.

Typed for typed/net/dns

```
(require typed/net/dns)
```
package: typed-racket-more

Typed for typed/net/ftp

```
(require typed/net/ftp)
```
package: typed-racket-more

FTP-Connection

Describes an open FTP connection.

Typed for typed/net/gifwrite

```
(require typed/net/gifwrite)
```
package: typed-racket-more

Typed for typed/net/git-checkout

```
(require typed/net/git-checkout)
```
package: typed-racket-more

Typed for typed/net/head

```
(require typed/net/head)
```
package: typed-racket-more

Typed for typed/net/http-client

```
(require typed/net/http-client)
```
package: typed-racket-more

HTTP-Connection

Describes an HTTP connection, corresponding to http-conn?.

Typed for typed/net/imap

```
(require typed/net/imap)
```
package: typed-racket-more

`IMAP-Connection`

Describes an IMAP connection.

Typed for typed/net/mime

```
(require typed/net/mime)
```
package: typed-racket-more

Typed for typed/net/nntp

```
(require typed/net/nntp)
```
package: typed-racket-more

Typed for typed/net/pop3

```
(require typed/net/pop3)
```
package: typed-racket-more

Typed for typed/net/qp

```
(require typed/net/qp)
```
package: typed-racket-more

Typed for typed/net/sendmail

```
(require typed/net/sendmail)
```
package: typed-racket-more

Typed for typed/net/sendurl

```
(require typed/net/sendurl)
```
package: typed-racket-more

Typed for typed/net/smtp

```
(require typed/net/smtp)
```
package: typed-racket-more

Typed for typed/net/uri-codec

```
(require typed/net/uri-codec)
```
package: typed-racket-more

Typed for typed/net/url-connect

```
(require typed/net/url-connect)
```
package: typed-racket-more

Typed for typed/net/url-structs

```
(require typed/net/url-structs)
```
package: typed-racket-more

Path/Param

Describes the path/param struct from net/url-structs.

URL

Describes an url struct from net/url-structs.

Typed for typed/net/url

```
(require typed/net/url)
```
package: typed-racket-more

In addition to defining the following types, this module also provides the HTTP-Connection type defined by typed/net/http-client, and the URL and Path/Param types from typed/net/url-structs.

URL-Exception

Describes exceptions raised by URL-related functions; corresponds to url-exception?.

PortT

Describes the functions head-pure-port, delete-pure-port, get-impure-port, head-impure-port, and delete-impure-port.

PortT/Bytes

Like PortT, but describes the functions that make POST and PUT requests, which require an additional byte-string argument for POST or PUT data.

Typed for typed/openssl

```
(require typed/openssl)
```
package: typed-racket-more

SSL-Protocol

Describes an SSL protocol, is an alias for `(U 'auto 'sslv2-or-v3 'sslv2 'sslv3 'tls 'tls11 'tls12)`.

SSL-Server-Context
SSL-Client-Context

Describes an OpenSSL server or client context.

SSL-Context

Supertype of OpenSSL server and client contexts.

SSL-Listener

Describes an SSL listener, as produced by `ssl-listen`.

SSL-Verify-Source

Describes a verification source usable by `ssl-load-verify-source!` and the `ssl-default-verify-sources` parameter.

Typed for typed/openssl/md5

```
(require typed/openssl/md5)
```
package: typed-racket-more

Typed for typed/openssl/sha1

```
(require typed/openssl/sha1)
```
package: typed-racket-more

Typed for typed/racket/async-channel

```
(require typed/racket/async-channel)
```
package: typed-racket-more

Added in version 1.1 of package typed-racket-lib.

Typed for typed/racket/date

```
(require typed/racket/date)
```
package: typed-racket-lib

Typed for typed/racket/draw

```
(require typed/racket/draw)
```
package: typed-racket-more

```
LoadFileKind
```

Is an alias for (U 'unknown 'unknown/mask 'unknown/alpha 'gif 'gif/mask 'gif/alpha 'jpeg 'jpeg/alpha 'png 'png/mask 'png/alpha 'xbm 'xbm/alpha 'xpm 'xpm/alpha 'bmp 'bmp/alpha).

Typed for typed/racket/extflonum

```
(require typed/racket/extflonum)
```
package: typed-racket-more

```
(for/extflvector type-ann-maybe (for-clause ...) expr ...+)
(for*/extflvector type-ann-maybe (for-clause ...) expr ...+)
```

Typed for typed/racket/flonum

```
(require typed/racket/flonum)
```
package: typed-racket-more

```
(for/flvector maybe-length (for-clause ...) expr ...+)
(for*/flvector maybe-length (for-clause ...) expr ...+)
```

Typed for typed/racket/gui

```
(require typed/racket/gui)
```
package: typed-racket-more

```
Edit-Op
```

Is an alias for `(U 'undo 'redo 'clear 'cut 'copy 'paste 'kill 'select-all 'insert-text-box 'insert-pasteboard-box 'insert-image)`.

```
Read/Write-Format
```

Is an alias for `(U 'standard 'text 'text-force-cr)`.

```
File-Format
```

Is an alias for `(U 'guess 'same 'copy Read/Write-Format)`.

```
Threshold
```

Is an alias for `(U 'no-caret 'show-inactive-caret 'show-caret)`.

```
Draw-Caret
```

Is an alias for `(U Threshold (Pairof Natural Natural))`.

Typed for typed/racket/gui/no-check

```
(require typed/racket/gui/no-check)
```
package: typed-racket-more

Typed for typed/racket/random

```
(require typed/racket/random)
```
package: typed-racket-more

Added in version 1.5 of package `typed-racket-lib`.

Typed for typed/racket/sandbox

```
(require typed/racket/sandbox)
```
package: typed-racket-more

Typed for typed/racket/snip

```
(require typed/racket/snip)
```
package: typed-racket-more

Image-Kind

Is an alias for (U 'unknown 'unknown/mask 'unknown/alpha 'gif 'gif/mask 'gif/alpha 'jpeg 'png 'png/mask 'png/alpha 'xbm 'xpm 'bmp 'pict).

Typed for typed/racket/system

```
(require typed/racket/system)
```
package: typed-racket-lib

Typed for typed/rackunit/docs-complete

```
(require typed/rackunit/docs-complete)
```
package: rackunit-typed

Typed for typed/rackunit/gui

```
(require typed/rackunit/gui)
```
package: rackunit-typed

Typed for typed/rackunit/text-ui

```
(require typed/rackunit/text-ui)
```
package: rackunit-typed

Typed for typed/rackunit

```
(require typed/rackunit)
```
package: rackunit-typed

Typed for typed/srfi/14

```
(require typed/srfi/14)
```
package: typed-racket-more

Char-Set

Describes a character set usable by the srfi/14 functions.

Cursor

Describes a cursor for iterating over character sets.

Typed for typed/srfi/19

(require typed/srfi/19) package: typed-racket-more

Time
Date

Describes an SRFI 19 time or date structure.

Typed for typed/syntax/stx

(require typed/syntax/stx) package: typed-racket-more

Typed for typed/web-server/configuration/responders

(require typed/web-server/configuration/responders)
package: typed-racket-more

Typed for typed/web-server/http

(require typed/web-server/http) package: typed-racket-more

Changed in version 1.10 of package typed-racket-lib: Updated to reflect web-server/http version 1.3.
Changed in version 1.11: Updated to reflect web-server/http version 1.4.
Changed in version 1.13: Updated to reflect web-server/http version 1.6.

Typed for typed/db

(require typed/db) package: typed-racket-more

Typed for typed/db/base

```
(require typed/db/base)
```
package: typed-racket-more

Typed for typed/db/sqlite3

```
(require typed/db/sqlite3)
```
package: typed-racket-more

In some cases, these typed adapters may not contain all of exports of the original module, or their types may be more limited.

Other libraries included in the main distribution that are either written in Typed Racket or have adapter modules that are typed:

```
(require math)
```
package: math-lib

```
(require plot)
```
package: plot-gui-lib

Typed for typed/pict

```
(require typed/pict)
```
package: typed-racket-more

```
(require images/flomap)
```
package: images-lib

Typed for typed/images/logos

```
(require typed/images/logos)
```
package: typed-racket-more

Typed for typed/images/icons

```
(require typed/images/icons)
```
package: typed-racket-more

Typed for typed/images/compile-time

```
(require typed/images/compile-time)
```
package: typed-racket-more

3.1 Porting Untyped Modules to Typed Racket

To adapt a Racket library not included with Typed Racket, the following steps are required:

- Determine the data manipulated by the library, and how it will be represented in Typed Racket.
- Specify that data in Typed Racket, using `require/typed` and `#:opaque` and/or `#:struct`.
- Use the data types to import the various functions and constants of the library.
- Provide all the relevant identifiers from the new adapter module.

For example, the following module adapts the untyped `racket/bool` library:

```
#lang typed/racket
(require/typed racket/bool
               [true Boolean]
               [false Boolean]
               [symbol=? (Symbol Symbol -> Boolean)]
               [boolean=? (Boolean Boolean -> Boolean)]
               [false? (Any -> Boolean)])
(provide true false symbol=? boolean=? false?)
```

More substantial examples are available in the `typed` collection.

4 Typed Classes

Warning: the features described in this section are experimental and may not work correctly. Some of the features will change by the next release. In particular, typed-untyped interaction for classes will not be backwards compatible so do not rely on the current semantics.

Typed Racket provides support for object-oriented programming with the classes and objects provided by the `racket/class` library.

4.1 Special forms

`(require typed/racket/class)` package: `typed-racket-lib`

The special forms below are provided by the `typed/racket/class` and `typed/racket` modules but not by `typed/racket/base`. The `typed/racket/class` module additional provides all other bindings from `racket/class`.

```
(class superclass-expr
  maybe-type-parameters
  class-clause ...)
```

```
        class-clause = (inspect inspector-expr)
                     | (init init-decl ...)
                     | (init-field init-decl ...)
                     | (init-rest id/type)
                     | (field field-decl ...)
                     | (inherit-field field-decl ...)
                     | (public maybe-renamed/type ...)
                     | (pubment maybe-renamed/type ...)
                     | (override maybe-renamed/type ...)
                     | (augment maybe-renamed/type ...)
                     | (private id/type ...)
                     | (inherit id ...)
                     | method-definition
                     | definition
                     | expr
                     | (begin class-clause ...)

maybe-type-parameters =
                     | #:forall (type-variable ...)
                     | #:∀ (type-variable ...)

           init-decl = id/type
                     | [renamed]
                     | [renamed : type-expr]
                     | [maybe-renamed default-value-expr]
                     | [maybe-renamed : type-expr default-value-expr]

          field-decl = (maybe-renamed default-value-expr)
                     | (maybe-renamed : type-expr default-value-expr)

             id/type = id
                     | [id : type-expr]

  maybe-renamed/type = maybe-renamed
                     | [maybe-renamed : type-expr]

       maybe-renamed = id
                     | renamed

             renamed = (internal-id external-id)
```

Produces a class with type annotations that allows Typed Racket to type-check the methods, fields, and other clauses in the class.

The meaning of the class clauses are the same as in the `class` form from the `racket/class` library with the exception of the additional optional type annotations. Additional class clause

forms from `class` that are not listed in the grammar above are not currently supported in Typed Racket.

Examples:

```
> (define fish%
    (class object%
      (init [size : Real])

      (: current-size Real)
      (define current-size size)

      (super-new)

      (: get-size (-> Real))
      (define/public (get-size)
        current-size)

      (: grow (Real -> Void))
      (define/public (grow amt)
        (set! current-size (+ amt current-size)))

      (: eat ((Object [get-size (-> Real)]) -> Void))
      (define/public (eat other-fish)
        (grow (send other-fish get-size)))))
> (define dory (new fish% [size 5.5]))
```

Within a typed class form, one of the class clauses must be a call to `super-new`. Failure to call `super-new` will result in a type error. In addition, dynamic uses of `super-new` (e.g., calling it in a separate function within the dynamic extent of the class form's clauses) are restricted.

Example:

```
> (class object%
    ; Note the missing `super-new`
    (init-field [x : Real 0] [y : Real 0]))
racket/collects/racket/private/class-undef.rkt:46:6: Type
Checker: ill-formed typed class;
 must call `super-new' at the top-level of the class
  in: (#%expression (#%app compose-class (quote eval:4:0)
object% (#%app list) (#%app current-inspector) (quote #f)
(quote #f) (quote 2) (quote (x y)) (quote ()) (quote ())
(quote ()) (quote ()) (quote ()) (quote ()) (quote ())
(quote ()) (quote ()) (quote ()...
```

If any identifier with an optional type annotation is left without an annotation, the type-

checker will assume the type `Any` (or `Procedure` for methods) for that identifier.

Examples:

```
> (define point%
    (class object%
      (super-new)
      (init-field x y)))
> point%
- : (Class (init (x Any) (y Any)) (field (x Any) (y Any)))
#<class:point%>
```

When *type-variable* is provided, the class is parameterized over the given type variables. These type variables are in scope inside the body of the class. The resulting class can be instantiated at particular types using `inst`.

Examples:

```
> (define cons%
    (class object%
      #:forall (X Y)
      (super-new)
      (init-field [car : X] [cdr : Y])))
> cons%
- : (All (X Y) (Class (init (car X) (cdr Y)) (field (car X) (cdr
Y))))
#<class:cons%>
> (new (inst cons% Integer String) [car 5] [cdr "foo"])
- : (Object (field (car Integer) (cdr String)))
(object:cons% ...)
```

Initialization arguments may be provided by-name using the `new` form, by-position using the `make-object` form, or both using the `instantiate` form.

As in ordinary Racket classes, the order in which initialization arguments are declared determines the order of initialization types in the class type.

Furthermore, a class may also have a typed `init-rest` clause, in which case the class constructor takes an unbounded number of arguments by-position. The type of the `init-rest` clause must be either a `List` type, `Listof` type, or any other list type.

Examples:

```
> (define point-copy%
    ; a point% with a copy constructor
    (class object%
```

```
      (super-new)
      (init-rest [rst : (U (List Integer Integer)
                           (List (Object (field [x Integer]
                                                [y Integer]))))])
      (field [x : Integer 0] [y : Integer 0])
      (match rst
        [(list (? integer? *x) *y)
         (set! x *x) (set! y *y)]
        [(list (? (negate integer?) obj))
         (set! x (get-field x obj))
         (set! y (get-field y obj))])))
> (define p1 (make-object point-copy% 1 2))
> (make-object point-copy% p1)
- : (Object (field (x Integer) (y Integer)))
(object:point-copy% ...)
```

```
(define/public id expr)
(define/public (id . formals) body ...+)
```

Like `define/public` from `racket/class`, but uses the binding of `define` from Typed Racket.

The `formals` may specify type annotations as in `define`.

```
(define/override id expr)
(define/override (id . formals) body ...+)
```

Like `define/override` from `racket/class`, but uses the binding of `define` from Typed Racket.

The `formals` may specify type annotations as in `define`.

```
(define/pubment id expr)
(define/pubment (id . formals) body ...+)
```

Like `define/pubment` from `racket/class`, but uses the binding of `define` from Typed Racket.

The `formals` may specify type annotations as in `define`.

```
(define/augment id expr)
(define/augment (id . formals) body ...+)
```

Like `define/augment` from `racket/class`, but uses the binding of `define` from Typed Racket.

The `formals` may specify type annotations as in `define`.

```
(define/private id expr)
(define/private (id . formals) body ...+)
```

Like `define/private` from `racket/class`, but uses the binding of `define` from Typed Racket.

The `formals` may specify type annotations as in `define`.

```
(init init-decl ...)
(init-field init-decl ...)
(field field-decl ...)
(inherit-field field-decl ...)
(init-rest id/type)
(public maybe-renamed/type ...)
(pubment maybe-renamed/type ...)
(override maybe-renamed/type ...)
(augment maybe-renamed/type ...)
(private id/type ...)
(inherit maybe-renamed/type ...)
```

These forms are mostly equivalent to the forms of the same names from the `racket/class` library and will expand to them. However, they also allow the initialization argument, field, or method names to be annotated with types as described above for the `class` form.

4.2 Types

```
(Class class-type-clause ...)

class-type-clause = name+type
                  | (init init-type ...)
                  | (init-field init-type ...)
                  | (init-rest name+type)
                  | (field name+type ...)
                  | (augment name+type ...)
                  | #:implements type-alias-id
                  | #:implements/inits inits-id
                  | #:row-var row-var-id

        init-type = name+type
                  | [id type #:optional]

        name+type = [id type]
```

The type of a class with the given initialization argument, method, and field types.

Example:

```
> (: food% (Class (init [liquid? Boolean])
                  (field [nutrition Integer])
                  [get-nutrition (-> Integer)]))
```

The types of methods are provided either without a keyword, in which case they correspond to public methods, or with the `augment` keyword, in which case they correspond to a method that can be augmented.

An initialization argument type specifies a name and type and optionally a `#:optional` keyword. An initialization argument type with `#:optional` corresponds to an argument that does not need to be provided at object instantiation.

Example:

```
> (: drink% (Class (init [color String]
                         [carbonated? Boolean]
                         [viscosity Positive-Real #:optional])))
```

The order of initialization arguments in the type is significant, because it determines the types of by-position arguments for use with `make-object` and `instantiate`. A given `Class` type may also only contain a single `init-rest` clause.

Examples:

```
> (define drink%
    (class object%
      (super-new)
      ; The order of `color' and `carbonated?' cannot be swapped
      (init color carbonated? [viscosity 1.002])))
; The order of initialization matches the order in the type
> (make-object drink% "purple" #t)
- : (Object)
(object:drink% ...)
```

When *`type-alias-id`* is provided, the resulting class type includes all of the method and field types from the specified type alias (which must be an alias for a class type). This is intended to allow a type for a subclass to include parts of its parent class type. The initialization argument types of the parent, however, are *not* included because a subclass does not necessarily share the same initialization arguments as its parent class.

Initialization argument types can be included from the parent by providing *`inits-id`* with the `#:implements/inits` keyword. This is identical to the `#:implements` clause except

for the initialization argument behavior. Only a single `#:implements/inits` clause may be provided for a single `Class` type. The initialization arguments copied from the parent type are appended to the initialization arguments specified via the `init` and `init-field` clauses.

Multiple `#:implements` clauses may be provided for a single class type. The types for the `#:implements` clauses are merged in order and the last type for a given method name or field is used (the types in the `Class` type itself takes precedence).

Examples:

```
> (define-type Point<%> (Class (field [x Real] [y Real])))
> (: colored-point% (Class #:implements Point<%>
                           (field [color String])))
```

When *row-var-id* is provided, the class type is an abstract type that is row polymorphic. A row polymorphic class type can be instantiated at a specific row using `inst`. Only a single `#:row-var` clause may appear in a class type.

```
ClassTop
```

The supertype of all class types. A value of this type cannot be used for subclassing, object creation, or most other class functions. Its primary use is for reflective operations such as `is-a?`.

```
(Object object-type-clause ...)

object-type-clause = name+type
                   | (field name+type ...)
```

The type of an object with the given field and method types.

Examples:

```
> (new object%)
- : (Object)
(object)
> (new (class object% (super-new) (field [x : Real 0])))
- : (Object (field (x Real)))
(object:eval:20:0 ...)
```

```
(Instance class-type-expr)
```

The type of an object that corresponds to *class-type-expr*.

This is the same as an `Object` type that has all of the method and field types from *class-type-expr*. The types for the `augment` and `init` clauses in the class type are ignored.

Examples:

```
> (define-type Point% (Class (init-field [x Integer] [y Integer])))
> (: a-point (Instance Point%))
> (define a-point
    (new (class object%
           (super-new)
           (init-field [x : Integer 0] [y : Integer 0]))))
```

```
(Row class-type-clause ...)
```

Represents a row, which is used for instantiating row-polymorphic function types. Accepts all clauses that the `Class` form accepts except the keyword arguments.

Rows are not types, and therefore cannot be used in any context except in the `row-inst` form. See `row-inst` for examples.

5 Typed Units

Warning: the features described in this section are experimental and may not work correctly. Some of the features may change by the next release.

Typed Racket provides support for modular programming with the units and signatures provided by the `racket/unit` library.

5.1 Special forms

`(require typed/racket/unit)` package: `typed-racket-lib`

The special forms below are provided by the `typed/racket/unit` and `typed/racket` modules, but not by `typed/racket/base`. The `typed/racket/unit` module additionally provides all other bindings from `racket/unit`.

```
(define-signature id extension-decl
  (sig-elem ...))

extension-decl =
               | extends sig-id

     sig-elem = [id : type]
```

Binds an identifier to a signature and registers the identifier in the signature environment with the specified type bindings. Sigantures in Typed Racket allow only specifications of variables and their types. Variable and syntax definitions are not allowed in the `define-signature` form. This is only a limitation of the `define-signature` form in Typed Racket.

As in untyped Racket, the `extends` clause includes all elements of extended signature and any implementation of the new signature can be used as an implementation of the extended signature.

```
(unit
  (import sig-spec ...)
  (export sig-spec ...)
  init-depends-decl
  unit-body-expr-or-defn
  ...)
```

```
        sig-spec = sig-id
                 | (prefix id sig-spec)
                 | (rename sig-spec (id id) ...)
                 | (only sig-spec id ...)
                 | (except sig-spec id ...)

init-depends-decl =
                 | (init-depend sig-id ...)
```

The typed version of the Racket `unit` form. Unit expressions in Typed Racket do not support tagged signatures with the `tag` keyword.

```
(invoke-unit unit-expr)
(invoke-unit unit-expr (import sig-spec ...))
```

The typed version of the Racket `invoke-unit` form.

```
(define-values/invoke-unit unit-expr
  (import def-sig-spec ...)
  (export def-sig-spec ...))

def-sig-spec = sig-id
             | (prefix id def-sig-spec)
             | (rename def-sig-spec (id id) ...)
```

The typed version of the Racket `define-values/invoke-unit` form. In Typed Racket `define-values/invoke-unit` is only allowed at the top-level of a module.

```
(compound-unit
  (import link-binding ...)
  (export link-id ...)
  (link linkage-decl ...))

link-binding = (link-id : sig-id)

linkage-decl = ((link-binding ...) unit-expr link-id ...)
```

The typed version of the Racket `compound-unit` form.

```
(define-unit unit-id
  (import sig-spec ...)
  (export sig-spec ...)
  init-depends-decl
  unit-body-expr-or-defn
  ...)
```

The typed version of the Racket `define-unit` form.

```
(compound-unit/infer
  (import infer-link-import ...)
  (export infer-link-export ...)
  (link infer-linkage-decl ...))

 infer-link-import = sig-id
                   | (link-id : sig-id)

 infer-link-export = link-id
                   | sig-id

infer-linkage-decl = ((link-binding ...) unit-id
                                         tagged-link-id ...)
                   | unit-id
```

The typed version of the Racket `compound-unit/infer` form.

```
(define-compound-unit id
  (import link-binding ...)
  (export link-id ...)
  (link linkage-decl ...))
```

The typed version of the Racket `define-compound-unit` form.

```
(define-compound-unit/infer id
  (import link-binding ...)
  (export infer-link-export ...)
  (link infer-linkage-decl ...))
```

The typed version of the Racket `define-compound-unit/infer` form.

```
(invoke-unit/infer unit-spec)

unit-spec = unit-id
          | (link link-unit-id ...)
```

The typed version of the Racket `invoke-unit/infer` form.

```
(define-values/invoke-unit/infer maybe-exports unit-spec)

maybe-exports =
              | (export sig-sepc ...)

    unit-spec = unit-id
              | (link link-unit-id ...)
```

The typed version of the Racket `define-values/invoke-unit/infer` form. Like the `define-values/invoke-unit` form above, this form is only allowed at the toplevel of a module.

```
(unit-from-context sig-spec)
```

The typed version of the Racket `unit-from-context` form.

```
(define-unit-from-context id sig-spec)
```

The typed version of the Racket `define-unit-from-context` form.

5.2 Types

```
(Unit
  (import sig-id ...)
  (export sig-id ...)
  optional-init-depend-clause
  optional-body-type-clause)

optional-init-depend-clause =
                            | (init-depend sig-id ...)

  optional-body-type-clause =
                            | type
                            | (Values type ...)
```

The type of a unit with the given imports, exports, initialization dependencies, and body type. Omitting the init-depend clause is equivalent to an `init-depend` clause that contains no signatures. The body type is the type of the last expression in the unit's body. If a unit contains only definitions and no expressions its body type is `Void`. Omitting the body type is equivalent to specifying a body type of `Void`.

Example:

```
> (module Unit-Types typed/racket
    (define-signature fact^ ([fact : (-> Natural Natural)]))
    (: use-fact@ (Unit (import fact^)
                       (export)
                       Natural))
    (define use-fact@ (unit (import fact^) (export) (fact 5))))
```

```
UnitTop
```

The supertype of all unit types. Values of this type cannot be linked or invoked. The primary use of is for the reflective operation `unit?`

5.3 Interacting with Untyped Code

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

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

maybe-renamed = id
              | (orig-id new-id)

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

The `#:signature` clause of `require/typed` requires the given signature and registers it in the signature environment with the specified bindings. Unlike other identifiers required with `require/typed`, signatures are not protected by contracts.

Signatures are not runtime values and therefore do not need to be protected by contracts.

Examples:

```
> (module UNTYPED-1 racket
    (provide a^)
    (define-signature a^ (a)))
> (module TYPED-1 typed/racket
    (require/typed 'UNTYPED-1
                   [#:signature a^ ([a : Integer])])
    (unit (import a^) (export) (add1 a)))
```

Typed Racket will infer whether the named signature `extends` another signature. It is an error to require a signature that extends a signature not present in the signature environment.

Examples:

```
> (module UNTYPED-2 racket
    (provide a-sub^)
    (define-signature a^ (a1))
    (define-signature a-sub^ extends a^ (a2)))
> (module TYPED-2 typed/racket
    (require/typed 'UNTYPED-2
                   [#:signature a-sub^
                     ([a1 : Integer]
                      [a2 : String])]))
```

```
eval:6:0: Type Checker: Error in macro expansion -- required
signature extends an untyped signature
  required signature: a-sub^
  extended signature: a^
  in: UNTYPED-2
```

Requiring a signature from an untyped module that contains variable definitions is an error in Typed Racket.

Examples:

```
> (module UNTYPED racket
    (provide bad^)
    (define-signature bad^ (bad (define-values (bad-
ref) (car bad)))))
> (module TYPED typed/racket
    (require/typed 'UNTYPED
                   [#:signature bad^
                     ([bad : (Pairof Integer Integer)]
                      [bad-ref : Integer])]))
eval:8:0: Type Checker: Error in macro expansion -- untyped
signatures containing definitions are prohibited
  in: UNTYPED
```

5.4 Limitations

5.4.1 Signature Forms

Unlike Racket's `define-signature` form, in Typed Racket `define-signature` only supports one kind of signature element that specifies the types of variables in the signature. In particular Typed Racket's `define-signature` form does not support uses of `define-syntaxes`, `define-values`, or `define-values-for-export` . Requiring an untyped signature that contains definitions in a typed module will result in an error.

Examples:

```
> (module UNTYPED racket
    (provide bad^)
    (define-signature bad^ ((define-values (bad) 13))))
> (module TYPED typed/racket
    (require/typed 'UNTYPED
                   [#:signature bad^ ([bad : Integer])]))
eval:10:0: Type Checker: Error in macro expansion -- untyped
signatures containing definitions are prohibited
  in: UNTYPED
```

5.4.2 Contracts and Unit Static Information

Unit values that flow between typed and untyped contexts are wrapped in `unit/c` contracts to guard the unit's imports, exports, and result upon invocation. When identifers that are additionally bound to static information about a unit, such as those defined by `define-unit`, flow between typed and untyped contexts contract application can result the static information becoming inaccessible.

Examples:

```
> (module UNTYPED racket
    (provide u@)
    (define-unit u@ (import) (export) "Hello!"))
> (module TYPED typed/racket
    (require/typed 'UNTYPED
                   [u@ (Unit (import) (export) String)])
    (invoke-unit/infer u@))
eval:12:0: untyped-invoke-unit/infer: unknown unit
definition
  at: u@
  in: (untyped-invoke-unit/infer u@)
```

When an identifier bound to static unit information flows from a typed module to an untyped module, however, the situation is worse. Because unit static information is bound to an identifier as a macro definition, any use of the typed unit is disallowed in untyped contexts.

Examples:

```
> (module TYPED typed/racket
    (provide u@)
    (define-unit u@ (import) (export) "Hello!"))
> (module UNTYPED racket
    (require 'TYPED)
    u@)
eval:14:0: Type Checker: Macro u@ from typed module used in
untyped code
  in: u@
```

5.4.3 Signatures and Internal Definition Contexts

Typed Racket's `define-signature` form is allowed in both top-level and internal definition contexts. As the following example shows, defining signatures in internal definiition contexts can be problematic.

Example:

```
> (module TYPED typed/racket
    (define-signature a^ ())
    (define u@
      (let ()
        (define-signature a^ ())
        (unit (import a^) (export) (init-depend a^) 5)))
    (invoke-unit u@ (import a^)))
eval:15:0: Type Checker: type mismatch
  expected: (Unit (import a^) (export) (init-depend a^)
AnyValues)
  given: (Unit (import a^) (export) (init-depend a^)
Positive-Byte)
  in: a^
```

Even though the unit imports a signature named `a^`, the `a^` provided for the import refers to the top-level `a^` signature and the type system prevents invoking the unit. This issue can be avoided by defining signatures only at the top-level of a module.

5.4.4 Tagged Signatures

Various unit forms in Racket allow for signatures to be tagged to support the definition of units that import or export the same signature multiple times. Typed Racket does not support the use of tagged signatures, using the `tag` keyword, anywhere in the various unit forms described above.

5.4.5 Structural Matching and Other Unit Forms

Typed Racket supports only those unit forms described above. All other bindings exported by `racket/unit` are not supported in the type system. In particular, the structural matching forms including `unit/new-import-export` and `unit/s` are unsupported.

6 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)
- : String
"7"
> (define: y : (U String Symbol) "hello")
> y
- : (U String Symbol)
"hello"
> (assert y string?)
- : String
"hello"
> (assert y boolean?)
Assertion #<procedure:boolean?> failed on "hello"
```

```
(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]))
eval:10:0: Type Checker: incomplete coverage; missing
coverage of Negative-Integer
  in: #f
```

```
(assert-typecheck-fail body-expr)
(assert-typecheck-fail body-expr #:result result-expr)
```

Explicitly produce a type error if *body-expr* does not produce a type error. If *result-expr* is provided, it will be the result of evaluating the expression, otherwise (void) will be returned. If there is an expected type, that type is propagated as the expected type when checking *body-expr*.

Added in version 1.7 of package typed-racket-lib.

6.1 Ignoring type information

In some contexts, it is useful to have the typechecker forget type information on particular expressions. Any expression with the shape `(#%expression sub)` that has a true value for the syntax property `'typed-racket:ignore-type-information` will have the type `Any`, and the type checker won't learn anything about the expression for use in refining other types.

Added in version 1.7 of package `typed-racket-lib`.

The expression `sub` must still type check, but can have any single-valued type.

This is similar to `(ann sub Any)`, but differs in whether the typechecker can use this to refine other types, and can be used in context that do not depend on Typed Racket.

6.2 Untyped Utilities

`(require typed/untyped-utils)` package: `typed-racket-lib`

These utilities help interface typed with untyped code, particularly typed libraries that use types that cannot be converted into contracts, or export syntax transformers that must expand differently in typed and untyped contexts.

Changed in version 1.14 of package `typed-racket-lib`: The module moved from `typed-racket-more` to `typed-racket-lib`.

```
(require/untyped-contract maybe-begin module [name subtype] ...)

maybe-begin =
            | (begin expr ...)
```

Use this form to import typed identifiers whose types cannot be converted into contracts, but have *subtypes* that can be converted into contracts.

For example, suppose we define and provide the Typed Racket function

```
(: negate (case-> (-> Index Fixnum)
                  (-> Integer Integer)))
(define (negate x) (- x))
```

Trying to use `negate` within an untyped module will raise an error because the cases cannot be distinguished by arity alone.

If the defining module for `negate` is `"my-numerics.rkt"`, it can be imported and used in untyped code this way:

```
(require/untyped-contract
 "my-numerics.rkt"
 [negate  (-> Integer Integer)])
```

The type `(-> Integer Integer)` is converted into the contract used for `negate`.

The `require/untyped-contract` form expands into a submodule with language `typed/racket/base`. Identifiers used in *subtype* expressions must be either in Typed Racket's base type environment (e.g. `Integer` and `Listof`) or defined by an expression in the *maybe-begin* form, which is spliced into the submodule. For example, the `math/matrix` module imports and reexports `matrix-expt`, which has a `case->` type, for untyped use in this way:

```
(provide matrix-expt)

(require/untyped-contract
 (begin (require "private/matrix/matrix-types.rkt"))
 "private/matrix/matrix-expt.rkt"
 [matrix-expt  ((Matrix Number) Integer -> (Matrix Number))])
```

The `(require "private/matrix/matrix-types.rkt")` expression imports the `Matrix` type.

If an identifier *name* is imported using `require/untyped-contract`, reexported, and imported into typed code, it has its original type, not *subtype*. In other words, *subtype* is used only to generate a contract for *name*, not to narrow its type.

Because of limitations in the macro expander, `require/untyped-contract` cannot currently be used in typed code.

```
(define-typed/untyped-identifier name typed-name untyped-name)
(define-typed/untyped-identifier name deep-name untyped-name shallow-
name optional-name)
```

Defines an identifier *name* that expands to one of the following identifiers depending on context. When two identifiers are provided, *name* expands to *typed-name* in typed contexts and to *untyped-name* in untyped contexts (more precisely, everywhere else). When four identifiers are provided, *name* expands to *deep-name* in Deep-typed contexts, to *untyped-name* in untyped contexts, to *shallow-name* in Shallow-typed contexts, and to *optional-name* in Optionally-typed contexts. §8 "Deep, Shallow, and Optional Semantics" explains these different contexts.

Suppose we define and provide a Typed Racket function with this type:

```
(: my-filter (All (a) (-> (-> Any Any : a) (Listof Any) (Listof a))))
```

This type cannot be converted into a contract because it accepts a predicate. Worse, `require/untyped-contract` does not help because `(All (a) (-> (-> Any Any) (Listof Any) (Listof a)))` is not a subtype.

In this case, we might still provide `my-filter` to untyped code using

```
(provide my-filter)

(define-typed/untyped-identifier my-filter
  typed:my-filter
  untyped:my-filter)
```

where `typed:my-filter` is the original `my-filter`, but imported using `prefix-in`, and `untyped:my-filter` is either a Typed Racket implementation of it with type `(All (a) (-> (-> Any Any) (Listof Any) (Listof a)))` or an untyped Racket implementation.

Avoid this if possible. Use only in cases where a type has no subtype that can be converted to a contract; i.e. cases in which `require/untyped-contract` cannot be used.

`(syntax-local-typed-context?)` → `boolean?`

Returns `#t` if called while expanding code in a typed context; otherwise `#f`.

This is the nuclear option, provided because it is sometimes, but rarely, useful. Avoid.

7 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 maybe-verbose t)

maybe-verbose =
              | #:verbose
```

Prints the type `t`. If `t` is a type alias (e.g., `Number`), then it will be expanded to its representation when printing. Any further type aliases in the type named by `t` will remain unexpanded.

If `#:verbose` is provided, all type aliases are expanded in the printed type.

Examples:

```
> (:type Number)
(U Exact-Number Imaginary Inexact-Complex Real)
[can expand further: Exact-Number Inexact-Complex Imaginary Real]
> (:type Real)
(U Negative-Real Nonnegative-Real)
[can expand further: Negative-Real Nonnegative-Real]
> (:type #:verbose Number)
(U 0
   1
   Byte-Larger-Than-One
   Exact-Complex
   Exact-Imaginary
   Float-Complex
   Float-Imaginary
   Float-Nan
   Float-Negative-Zero
   Float-Positive-Zero
   Negative-Fixnum
   Negative-Float-No-NaN
   Negative-Integer-Not-Fixnum
   Negative-Rational-Not-Integer
   Negative-Single-Flonum-No-Nan
   Positive-Fixnum-Not-Index
   Positive-Float-No-NaN
   Positive-Index-Not-Byte
   Positive-Integer-Not-Fixnum
   Positive-Rational-Not-Integer
```

```
    Positive-Single-Flonum-No-Nan
    Single-Flonum-Complex
    Single-Flonum-Imaginary
    Single-Flonum-Nan
    Single-Flonum-Negative-Zero
    Single-Flonum-Positive-Zero)
```

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

Prints the type of `e`, which must be an expression. This prints the whole type, which can sometimes be quite large.

Examples:

```
> (:print-type (+ 1 2))
Positive-Index
> (:print-type map)
(All (c a b ...)
  (case->
   (-> (-> a c) (Pairof a (Listof a)) (Pairof c (Listof c)))
   (-> (-> a b ... b c) (Listof a) (Listof b) ... b (Listof c))))
```

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

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

Example:

```
> (:query-type/args + Integer Number)
(-> Integer Number Number)
```

```
(: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`.

Examples:

```
> (:query-type/result + Integer)
(-> Integer * Integer)
> (:query-type/result + Float)
(case->
 (-> Flonum Flonum * Flonum)
 (-> Real Real Flonum Real * Flonum)
 (-> Real Flonum Real * Flonum)
 (-> Flonum Real Real * Flonum))
```

```
(:kind e)
```

Prints the kind of a well-kinded type-level expression `e`. When `e` is a type, it prints `*`. When `e` is a type constructor, `->` following the open parenthesis in the printed result indicates `e` is productive and `-o` indicates otherwise.

Examples:

```
> (:kind Integer)
*
> (:kind Listof)
(-> * *)
> (:kind Pairof)
(-> * * *)
> (:kind U)
(-o * ... *)
```

Added in version 1.15 of package `typed-racket-lib`.

8 Deep, Shallow, and Optional Semantics

typed/racket/deep

`#lang typed/racket/deep` package: `typed-racket-lib`

typed/racket/base/deep

`#lang typed/racket/base/deep` package: `typed-racket-lib`

typed/racket/shallow

`#lang typed/racket/shallow` package: `typed-racket-lib`

typed/racket/base/shallow

`#lang typed/racket/base/shallow` package: `typed-racket-lib`

typed/racket/optional

`#lang typed/racket/optional` package: `typed-racket-lib`

typed/racket/base/optional

`#lang typed/racket/base/optional` package: `typed-racket-lib`

See also: §6 “Typed-Untyped Interaction” in the Typed Racket Guide.

Typed Racket allows the combination of both typed and untyped code in a single program. Untyped code can freely import typed identifiers. Typed code can import untyped identifiers by giving them types (via `require/typed`).

Allowing typed/untyped combinations raises questions about *whether* and *how* types should constrain the behavior of untyped code. On one hand, strong type constraints are useful because they can detect when a typed-untyped interaction goes wrong. On the other hand, constraints must be enforced with run-time checks, which affect run-time performance. Stronger constraints generally impose a higher performance cost.

By default, Typed Racket provides Deep types that strictly constrain the behavior of untyped code. But because these constraints can be expensive, Typed Racket offers two alternatives:

Shallow and Optional types. All three use the same static types and static checks, but they progressively weaken the run-time behavior of types.

- *Deep types* enforce strong, compositional guarantees. If a value is annotated with a Deep type, then all of its interactions with other code must match the type. For example, a value with the type `(Listof String)` must be a list that contains only strings; otherwise, Typed Racket raises an error.

  Available in: `typed/racket`, `typed/racket/base`, `typed/racket/deep`, and `typed/racket/base/deep`.

- *Shallow types* enforce the outer shape of values. For example, the Shallow type `(Listof String)` checks only for lists — it does not check whether the list elements are strings. This enforcement may seem weak at first glance, but Shallow types can work together to provide a decent safety net. If Shallow-typed code gets an element from a list and expects a `String`, then another check will make sure the element is really a string.

  Available in: `typed/racket/shallow`, and `typed/racket/base/shallow`.

- *Optional types* enforce nothing and add zero run-time cost. These types are useful for finding bugs in typed code at compile-time, but they cannot detect interaction errors at run-time.

  Available in: `typed/racket/optional`, and `typed/racket/base/optional`.

8.1 Example Interactions

The examples below show how Deep, Shallow, and Optional change the run-time behavior (or, the semantics) of types.

8.1.1 Checking Immutable Data: Importing a List

When typed code imports an untyped list:

- Deep types check each element of the list at the boundary to untyped code;
- Shallow types check for a list, and check elements when they are accessed; and
- Optional types check nothing.

The following examples import the function `string->list`, which returns a list of characters, and use an incorrect type that expects a list of strings. Both Deep and Shallow types catch the error at some point. Optional types do not catch the error.

Deep types prevent a list of characters from entering typed code with the type `(Listof String)`:

```
#lang typed/racket ; or #lang typed/racket/deep

(require/typed racket/base
  [string->list (-> String (Listof String))])


(string->list "racket")
string->list: broke its own contract
  promised: string?
  produced: #\r
  in: an element of
      the range of
      (-> any/c (listof string?))
  contract from: (interface for string->list)
  blaming: (interface for string->list)
   (assuming the contract is correct)
  at: eval:1:0
```

Shallow types allow a list of characters to have the type `(Listof String)`, but detect an error if typed code reads an element from the list:

```
#lang typed/racket/shallow

(require/typed racket/base
  [string->list (-> String (Listof String))])

(define lst (string->list "racket"))


(first lst)
shape-check: value does not match expected type
  value: #\r
  type: String
  lang: 'typed/racket/shallow
  src: '(eval 3 0 3 1)
```

Optional types do not detect any error in this example:

```
#lang typed/racket/optional

(require/typed racket/base
  [string->list (-> String (Listof String))])

(define lst (string->list "racket"))
```

```
(first lst)
- : String
#\r
```

8.1.2 Checking Mutable Data: Importing a Vector

When typed code imports an untyped vector:

- Deep types wrap the vector in a contract that checks future reads and writes;
- Shallow types check for a vector at the boundary, and check elements on demand (same as for lists); and
- Optional types check nothing.

The following example imports `make-vector` with an incorrect type that expects a vector of strings as its output. When `make-vector` returns a vector of numbers instead, both Deep and Shallow types catch the error when reading from the vector. Optional types do not catch the error.

Deep catches a bad vector element:

```
#lang typed/racket ; or #lang typed/racket/deep

(require/typed racket/base
  [make-vector (-> Integer (Vectorof String))])

(define vec (make-vector 10))


(vector-ref vec 0)
make-vector: broke its own contract
  promised: string?
  produced: 0
  in: an element of
      the range of
      (-> any/c (vectorof string?))
  contract from: (interface for make-vector)
  blaming: (interface for make-vector)
   (assuming the contract is correct)
  at: eval:3:0
```

Shallow catches a bad vector element:

```
#lang typed/racket/shallow

(require/typed racket/base
  [make-vector (-> Integer (Vectorof String))])

(define vec (make-vector 10))


(vector-ref vec 0)
shape-check: value does not match expected type
  value: 0
  type: String
  lang: 'typed/racket/shallow
  src: '(eval 6 0 6 1)
```

Optional does not catch a bad element:

```
#lang typed/racket/optional

(require/typed racket/base
  [make-vector (-> Integer (Vectorof String))])

(define vec (make-vector 10))


(vector-ref vec 0)
- : String
0
```

8.1.3 Checking Functions that Cross Multiple Boundaries

Deep types can detect some errors that Shallow types miss, especially when a program contains several modules. This is because every module in a program can trust that every Deep type is a true claim, but only the one module that defines a Shallow type can depend on the type. In short, Deep types are *permanent* whereas Shallow types are *temporary*.

The following example uses three modules to create a situation where Deep types catch an error that Shallow types miss. First, the untyped module `racket/base` provides the standard `string-length` function. Second, a typed `interface` module imports `string-length` with an incorrect type and reprovides with a new name: `strlen`. Third, a typed client module imports `strlen` with a correct type and calls it on a string.

Deep types raise an error when `strlen` is called because of the incorrect type in the interface:

```
#lang typed/racket ; or #lang typed/racket/deep
```

```
(module interface typed/racket
  (require/typed racket/base
    [string-length (-> String Void)])
  (define strlen string-length)
  (provide strlen))

(require/typed 'interface
  [strlen (-> String Natural)])


(strlen "racket")
```
string-length: broke its own contract
promised: void?
produced: 6
in: (-> any/c void?)
contract from: (interface for string-length)
blaming: (interface for string-length)
(assuming the contract is correct)
at: eval:6:0

Shallow types do not raise an error because the interface type is not enforced for the outer client module:

```
#lang typed/racket/shallow

(module interface typed/racket/shallow
  (require/typed racket/base
    [string-length (-> String Void)])
  (define strlen string-length)
  (provide strlen))

(require/typed 'interface
  [strlen (-> String Natural)])


(strlen "racket")
- : Integer [more precisely: Nonnegative-Integer]
6
```

Optional types do not raise an error either:

```
#lang typed/racket/optional

(module interface typed/racket/optional
  (require/typed racket/base
```

```
    [string-length (-> String Void)])
  (define strlen string-length)
  (provide strlen))

(require/typed 'interface
  [strlen (-> String Natural)])

(strlen "racket")
- : Integer [more precisely: Nonnegative-Integer]
6
```

8.2 Forms that Depend on the Behavior of Types

The following Typed Racket forms use types to create run-time checks. Consequently, their behavior changes depending on whether types are Deep, Shallow, or Optional.

Across these forms, the changes are roughly the same. Deep types get enforced as (higher-order) contracts, Shallow types get enforced as shape checks, and Optional types get enforced with nothing. The key point to understand is *which* types get enforced at run-time.

- `require/typed` imports bindings from another module and attaches types to the bindings. The attached types get enforced.
- `cast` assigns a type to an expression. The assigned type gets enforced.
- `with-type` creates a typed region in untyped code. Types at the boundary between this region and untyped code get enforced.

The following forms modify the contracts that Deep Typed Racket generates. Uses of these forms may need to change to accommodate Shallow and Optional clients.

- `require/untyped-contract` brings an identifier from Deep-typed code to untyped code using a subtype of its actual type. If the required identifier travels from untyped code to a Shallow or Optional client, this client must work with the subtype. A Deep client would be able to use the normal type.
- `define-typed/untyped-identifier` accepts four identifiers to fine-tune its behavior for Deep, untyped, Shallow, and Optional clients.

8.2.1 Example: Casts in Deep, Shallow, and Optional

To give one example of a form that depends on the behavior of types, `cast` checks full types in Deep mode, checks shapes in Shallow mode, and checks nothing in Optional mode.

Deep detects a bad cast:

```
; #lang typed/racket
; or #lang typed/racket/deep
> (cast (list 42) (Listof String))
(cast for #f): broke its own contract
  promised: string?
  produced: 42
  in: an element of
      (listof string?)
  contract from: cast
  blaming: cast
   (assuming the contract is correct)
  at: eval:9:0
```

Shallow allows one bad cast but detects a shape-level one:

```
; #lang typed/racket/shallow
> (cast (list 42) (Listof String))
- : (Listof String)
'(42)
> (cast (list 42) Number)
shape-check: value does not match expected type
  value: '(42)
  type: Number
  lang: 'typed/racket/shallow
  src: '(eval 11 0 11 1)
```

Optional lets any cast succeed:

```
; #lang typed/racket/optional
> (cast (list 42) (Listof String))
- : (Listof String)
'(42)
> (cast (list 42) Number)
- : Number
'(42)
```

8.3 How to Choose Between Deep, Shallow, and Optional

Deep, Shallow, and Optional types have complementary strengths and weaknesses. Deep types give strong type guarantees and enable full type-directed optimizations, but may pay a high cost at boundaries. In particular, the costs for higher-order types are high. Examples include `HashTable`, `->*`, and `Object`. Shallow types give weak guarantees, but come at a

lower cost. The cost is constant-time for many types, including `HashTable` and `->*`, and linear-time for a few others such as `U` and `Object`. Optional types give no guarantees, but come at no cost.

Based on these tradeoffs, this section offers some advice about when to choose one style over the others.

8.3.1 When to Use Deep Types

Deep types are best in the following situations:

- For large blocks of typed code, to take full advantage of type-directed optimizations within each block.
- For tightly-connected groups of typed modules, because Deep types pay no cost to interact with one another.
- For modules in which you want the types to be fully enforced, perhaps for predicting the behavior of typed-untyped interactions or for debugging.

8.3.2 When to Use Shallow Types

Shallow types are best in the following situations:

- For typed code that frequently interacts with untyped code, especially when it sends large immutable values or higher-order values (vectors, functions, etc.) across boundaries.
- For large blocks of typed code that primarily uses basic values (numbers, strings, etc.) or monomorphic data structures. In such cases, Shallow types get the full benefit of type-directed optimizations and few run-time costs.
- For boundaries where Deep enforcement (via contracts) is too restrictive. For example, Deep code can never call a function that has the type `Procedure`, but Shallow can after a cast.
- For boundaries where Deep cannot convert the types to contracts, such as for a higher-order syntax object such as `(Syntaxof (Boxof Real))`.

8.3.3 When to Use Optional Types

Optional types are best in the following situations:

- For typed-to-untyped migrations where performance needs to be predictable, because an Optionally-typed program behaves just like a Racket program that ignores all the types.
- For boundaries that neither Deep nor Shallow can express. For example, only Optional can use occurrence types at a boundary.
- For prototyping; that is, for testing whether an idea can type-check without testing whether it interacts well with untyped code.

8.3.4 General Tips

- Deep, Shallow, and Optional use the same compile-time type checks, so switching a module from one style to another is usually a one-line change (to the `#lang` line).
- When converting a Racket program to Typed Racket, try Deep types at first and change to Shallow if run-time performance becomes a bottleneck (or, if contract wrappers raise a correctness issue).

8.4 Related Gradual Typing Work

Shallow Typed Racket implements the *Transient* semantics for gradual languages [Programming-2022, PLDI-2022], which was invented by Michael M. Vitousek [RP:DLS-2014, RP:POPL-2017, RP:Vitousek-2019, RP:DLS-2019]. Transient protects typed code by rewriting it to defensively check the shape of values whenever it calls a function, reads from a data structure, or otherwise receives input that may have come from an untyped source. Because of the rewriting, Transient is able to enforce type soundness without higher-order contracts.

Deep Typed Racket implements the standard semantics for gradual languages, which is known variously as Guarded [RP:POPL-2017], Natural [TOPLAS-2009], and Behavioral [KafKa-2018]. This *Guarded* semantics eagerly checks untyped values when possible and otherwise creates wrappers to defer checks.

Typed Racket uses the names “Shallow” and “Deep” rather than “Transient” and “Guarded” to emphasize the guarantees that such types provide instead than the method used to implement these guarantees. Shallow types provide a type soundness guarantee; Deep types provide type soundness and complete monitoring [OOPSLA-2019].

Optional types are a widely-used approach to gradual typing, despite their unsound support for typed-untyped interactions. Optionally-typed languages include the following: TypeScript, Flow, mypy, and Typed Clojure [ESOP-2016, Bonnaire-Sergeant-2019].

9 Typed Racket Syntax Without Type Checking

`#lang typed/racket/no-check` package: `typed-racket-lib`
`#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")
```

10 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/region.rkt:764:62: contract violation
  expected: number?
  given: #f
  in: the 1st argument of
      (-> number? any)
  contract from: (region typed-region)
  blaming: top-level
   (assuming the contract is correct)
> (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 own contract
promised: string?
produced: 'hello
in: string?
contract from: top-level
blaming: top-level
(assuming the contract is correct)
at: eval:5:0
```
> (with-type ([fun (Number -> Number)]
              [val Number])
    (define (fun x) x)
    (define val 17))
> (fun val)
17
```

11 Optimization in Typed Racket

[1]

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

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 if the environment variable `PLT_TR_NO_OPTIMIZE` is set (to any value) or if the current code inspector (see §14.10 "Code Inspectors") is insufficiently powerful to access `racket/unsafe/ops`, for example when executing in a sandbox (see §14.12 "Sandboxed Evaluation"). This prevents untrusted code from accessing these operations by exploiting errors in the type system.

11.1 Contract Optimization

Typed Racket generates contracts for its exports to protect them against untyped code. By default, these contracts do not check that typed code obeys the types. If you want to generate contracts that check both sides equally (for analysis, for teaching, etc.) then set the environment variable `PLT_TR_NO_CONTRACT_OPTIMIZE` to any value and recompile.

[1] See §7 "Optimization in Typed Racket" in the guide for tips to get the most out of the optimizer.

12 Unsafe Typed Racket operations

```
(require typed/racket/unsafe)
```
package: typed-racket-lib

Warning: the operations documented in this section are *unsafe*, meaning that they can circumvent the invariants of the type system. Unless the `#:no-optimize` language option is used, this may result in unpredictable behavior and may even crash Typed Racket.

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

This form requires identifiers from the module *m* with the same import specifications as `require/typed`.

Unlike `require/typed`, this form is unsafe and will not generate contracts that correspond to the specified types to check that the values actually match their types.

Examples:

```
> (require typed/racket/unsafe)
; import with a bad type
> (unsafe-require/typed racket/base [values (-> String Integer)])
; unchecked call, the result type is wrong
> (values "foo")
- : Integer
"foo"
```

Added in version 1.3 of package `typed-racket-lib`.
Changed in version 1.6: Added support for struct type variables

```
(unsafe-provide provide-spec ...)
```

This form declares exports from a module with the same syntax as the `provide` form.

Unlike `provide`, this form is unsafe and Typed Racket will not generate any contracts that correspond to the specified types. This means that uses of the exports in other modules may circumvent the type system's invariants. In particular, one typed module may unsafely provide identifiers imported from another typed module.

Additionally, importing an identififer that is exported with `unsafe-provide` into another typed module, and then re-exporting it with `provide` will not cause contracts to be generated.

Uses of the provided identifiers in other typed modules are not affected by `unsafe-provide`—in these situations it behaves identically to `provide`. Furthermore, other typed modules that *use* a binding that is in an `unsafe-provide` will still have contracts generated as usual.

Examples:

```
> (module t typed/racket/base
    (require typed/racket/unsafe)
    (: f (-> Integer Integer))
    (define (f x) (add1 x))
    ; unsafe export, does not install checks
    (unsafe-provide f))
> (module u racket/base
    (require 't)
    ; bad call that's unchecked
    (f "foo"))
> (require 'u)
add1: contract violation
  expected: number?
  given: "foo"
```

Added in version 1.3 of package `typed-racket-lib`.
Changed in version 1.8: Added support for re-provided typed variables

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

Like `require/typed/provide` except that this form is unsafe and will not generate contracts that correspond to the specified types to check that the values actually match their types.

13 Legacy Forms

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

```
(lambda: formals maybe-ret . body)

maybe-ret =
          | : type

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

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:
```

An alias for `lambda:`.

```
(plambda: (a ...) formals maybe-ret . body)
(plambda: (a ... b ooo) formals maybe-ret . 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*.

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

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

A function with optional arguments.

```
(popt-lambda: (a ...) formals maybe-ret . body)
(popt-lambda: (a ... a ooo) formals maybe-ret . body)
```

A polymorphic function with optional arguments.

```
case-lambda:
```

An alias for `case-lambda`.

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

A polymorphic function of multiple arities.

```
(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] ...) : t0 . 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`. As with `let:`, the type annotation is optional.

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

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

Examples:

```
> (define: foo : Integer 10)
> (define: (A) mt-seq : (Sequenceof A) empty-sequence)
> (define: (add [first : Integer]
                [rest  : Integer]) : Integer
    (+ first rest))
> (define: (A) (poly-app [func : (A A -> A)]
                         [first : A]
                         [rest  : A]) : A
    (func first rest))
```

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

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

maybe-type-name =
                | #:type-name type-id
```

Equivalent to using `define-struct` to define a structure with the property `prop:procedure` supplied with the procedure `e` of type *`proc-t`*.

Changed in version 1.13 of package `typed-racket-lib`: Deprecated

Changed in version 1.4 of package `typed-racket-lib`: Added the `#:type-name` option.

`struct:`

An alias for `struct`.

`define-struct:`

An alias for `define-struct`.

`define-struct/exec:`

An alias for `define-struct/exec`.

`for:`

An alias for `for`.

```
for*/and:
for*/first:
for*/flvector:
for*/extflvector:
for*/fold:
for*/foldr:
for*/hash:
for*/hasheq:
for*/hasheqv:
for*/hashalw:
for*/last:
for*/list:
for*/lists:
for*/set:
for*/or:
for*/product:
for*/sum:
for*/vector:
for*:
for/and:
```

```
for/first:
for/flvector:
for/extflvector:
for/fold:
for/foldr:
for/hash:
for/hasheq:
for/hasheqv:
for/hashalw:
for/last:
for/list:
for/lists:
for/set:
for/or:
for/product:
for/sum:
for/vector:
```

Aliases for the same iteration forms without a `:`.

Changed in version 1.12 of package `typed-racket-lib`: Added `for/foldr:` and `for*/foldr:`.

```
do:
```

An alias for `do`.

```
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`.

14 Compatibility Languages

`#lang typed/scheme` package: `typed-racket-compatibility`
`#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`.

15 Experimental Features

These features are currently experimental and subject to change.

```
(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`. These predicate-based refinements are distinct from Typed Racket's more general `Refine` form.

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

Defines an executable structure.

```
(define-new-subtype name (constructor t))
```

Defines a new type *name* that is a subtype of *t*. The *constructor* is defined as a function that takes a value of type *t* and produces a value of the new type *name*. A `define-new-subtype` definition is only allowed at the top level of a file or module.

This is purely a type-level distinction, with no way to distinguish the new type from the base type at runtime. Predicates made by `make-predicate` won't be able to distinguish them properly, so they will return true for all values that the base type's predicate would return true for. This is usually not what you want, so you shouldn't use `make-predicate` with these types.

Examples:

```
> (module m typed/racket
    (provide Radians radians f)
    (define-new-subtype Radians (radians Real))
    (: f : [Radians -> Real])
    (define (f a)
      (sin a)))
> (require 'm)
> (radians 0)
- : Real [more precisely: Radians]
0
> (f (radians 0))
- : Real
0
```

15.1 Logical Refinements and Linear Integer Reasoning

Typed Racket allows types to be 'refined' or 'constrained' by logical propositions. These propositions can mention certain program terms, allowing a program's types to depend on the values of terms.

```
(Refine [id : type] proposition)

    proposition = Top
                | Bot
                | (: symbolic-object type)
                | (! symbolic-object type)
                | (and proposition ...)
                | (or proposition ...)
                | (when proposition proposition)
                | (unless proposition proposition)
                | (if proposition proposition proposition)
                | (linear-comp symbolic-object symbolic-object)

    linear-comp = <
                | <=
                | =
                | >=
                | >

symbolic-object = exact-integer
                | symbolic-path
                | (+ symbolic-object ...)
                | (- symbolic-object ...)
                | (* exact-integer symbolic-object)

  symbolic-path = id
                | (path-elem symbolic-path)

      path-elem = car
                | cdr
                | vector-length
```

`(Refine [v : t] p)` is a refinement of type `t` with logical proposition `p`, or in other words it describes any value `v` of type `t` for which the logical proposition `p` holds.

Example:

```
> (ann 42 (Refine [n : Integer] (= n 42)))
- : Integer [more precisely: (Refine (x0 : Integer) (= 42 x0))]
42
```

Note: The identifier in a refinement type is in scope inside the proposition, but not the type.

`(: o t)` used as a proposition holds when symbolic object `o` is of type `t`.

```
(! sym-obj type)
```

This is the dual of `(: o t)`, holding when `o` is not of type `t`.

Propositions can also describe linear inequalities (e.g. `(<= x 42)` holds when `x` is less than or equal to `42`), using any of the following relations: `<=`, `<`, `=`, `>=`, `>`.

The following logical combinators hold as one would expect depending on which of their subcomponents hold: `and`, `or`, `if`, `not`.

`(when p q)` is equivalent to `(or (not p) (and p q))`.

`(unless p q)` is equivalent to `(or p q)`.

In addition to reasoning about propositions regarding types (i.e. something is or is not of some particular type), Typed Racket is equipped with a linear integer arithmetic solver that can prove linear constraints when necessary. To turn on this solver (and some other refinement reasoning), you must add the `#:with-refinements` keyword when specifying the language of your program:

```
#lang typed/racket #:with-refinements
```

With this language option on, type checking the following primitives will produce more specific logical info (when they are being applied to 2 or 3 arguments): `*`, `+`, `-`, `<`, `<=`, `=`, `>=`, `>`, and `make-vector`.

This allows code such as the following to type check:

```
(if (< 5 4)
    (+ "Luke," "I am your father")
    "that's impossible!")
```

i.e. with refinement reasoning enabled, Typed Racket detects that the comparison is guaranteed to produce `#f`, and thus the clearly ill-typed 'then'-branch is ignored by the type checker since it is guaranteed to be dead code.

15.2 Dependent Function Types

Typed Racket supports explicitly dependent function types:

```
(-> ([id : opt-deps arg-type] ...)
    opt-pre
    range-type
    opt-props)

    opt-deps =
             | (id ...)

     opt-pre =
             | #:pre (id ...) prop

   opt-props =
             | opt-pos-prop opt-neg-prop opt-obj

opt-pos-prop =
             | #:+ prop

opt-neg-prop =
             | #:- prop

     opt-obj =
             | #:object obj
```

The syntax is similar to Racket's dependent contracts syntax (i.e. `->i`).

Each function argument has a name, an optional list of identifiers it depends on, an argument type. An argument's type can mention (i.e. depend on) other arguments by name if they appear in its list of dependencies. Dependencies cannot be cyclic.

A function may have also have a precondition. The precondition is introduced with the `#:pre` keyword followed by the list of arguments on which it depends and the proposition which describes the precondition.

A function's range may depend on any of its arguments.

The grammar of supported propositions and symbolic objects (i.e. `prop` and `obj`) is the same as the `proposition` and `symbolic-object` grammars from `Refine`'s syntax.

For example, here is a dependently typed version of Racket's `vector-ref` which eliminates vector bounds errors during type checking instead of at run time:

```
> (require racket/unsafe/ops)
> (: safe-ref1 (All (A) (-> ([v : (Vectorof A)]
                             [n : (v) (Refine [i : Natural]
                                              (< i (vector-
length v)))])
                           A)))
```

```
> (define (safe-ref1 v n) (unsafe-vector-ref v n))
> (safe-ref1 (vector "safe!") 0)
- : String
"safe!"
> (safe-ref1 (vector "not safe!") 1)
eval:10:0: Type Checker: Polymorphic function `safe-ref1'
could not be applied to arguments:
Argument x0 (position 1):
   Expected: (Vectorof A)
   Given:       (Mutable-Vector String)
Argument y0 (position 2):
   Expected: (Refine (z0 : Nonnegative-Integer) (< z0
(vector-length x0)))
   Given:       (Refine (z0 : One) (= 1 z0))

   in: 1
```

Here is an equivalent type that uses a precondition instead of a refinement type:

```
> (: safe-ref2 (All (A) (-> ([v : (Vectorof A)]
                             [n : Natural])
                            #:pre (v n) (< n (vector-length v))
                            A)))
> (define (safe-ref2 v n) (unsafe-vector-ref v n))
> (safe-ref2 (vector "safe!") 0)
- : String
"safe!"
> (safe-ref2 (vector "not safe!") 1)
eval:14:0: Type Checker: could not apply function;
 unable to prove
   precondition: (<= 2 (vector-length a))
   in: 1
```

Using preconditions can provide more detailed type checker error messages, i.e. they can indicate when the arguments were of the correct type but the precondition could not be proven.

Bibliography

[DLS-2006] Sam Tobin-Hochstadt and Matthias Felleisen, "Interlanguage Migration: from Scripts to Programs," Dynamic Languages Symposium, 2006. https://www2.ccs.neu.edu/racket/pubs/dls06-tf.pdfPresents the original model for module-level gradual typing. In the model, one typed module may interact with any number of untyped modules. A type soundness theorem guarantees the integrity of all typed code.

[SFP-2007] Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt, "Advanced Macrology and the Implementation of Typed Scheme," Workshop on Scheme and Functional Programming, 2007. https://www2.ccs.neu.edu/racket/pubs/scheme2007-ctf.pdfDescribes the key macros that enabled Typed Racket.

[POPL-2008] Sam Tobin-Hochstadt and Matthias Felleisen, "The Design and Implementation of Typed Scheme," Symposium on Principles of Programming Languages, 2008. https://www2.ccs.neu.edu/racket/pubs/popl08-thf.pdfContains a model of core Typed Racket (with a simple form of occurrence typing) and an extended discussion about scaling the model to a language.

[ESOP-2009] T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen, "Practical Variable-Arity Polymorphism," European Symposium on Programming, 2009. https://www2.ccs.neu.edu/racket/pubs/esop09-sthf.pdfExplains how to type-check a polymorphic function that accepts any number of arguments (such as map).

[TOPLAS-2009] Jacob Matthews and Robert Bruce Findler, "Operational Semantics for Multi-Language Programs," ACM Transactions on Programming Languages and Systems, 2009. https://users.cs.northwestern.edu/~robby/pubs/papers/toplas09-mf.pdf

[ICFP-2010] Sam Tobin-Hochstadt and Matthias Felleisen, "Logical Types for Untyped Languages," International Conference on Functional Programming, 2010. https://www2.ccs.neu.edu/racket/pubs/icfp10-thf.pdfPresents a compositionas occurrence typing system and comments on its implementation in Typed Racket.

[Tobin-Hochstadt] Sam Tobin-Hochstadt, "Typed Scheme: From Scripts to Programs," Ph.D. dissertation, 2010. https://www2.ccs.neu.edu/racket/pubs/dissertation-tobin-hochstadt.pdf

[PLDI-2011] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen, "Languages as Libraries," Conference on Programming Language Design and Implementation, 2011. https://www2.ccs.neu.edu/racket/pubs/pldi11-thacff.pdfMotivates the use of macros to define a language and summarizes the Typed Racket type checker and optimizer.

[OOPSLA-2012] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen, "Gradual Typing for First-Class Classes," Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2012. https://www2.ccs.neu.edu/racket/pubs/oopsla12-tsdthf.pdfPresents a model of typed classes that can interact with untyped classes through method calls, inheritance, and mixins.

[PADL-2012] Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen, "Typing the Numeric Tower," International Symposium on Practical Aspects of Declarative Languages, 2012. https://www2.ccs.neu.edu/racket/pubs/padl12-stff.pdfMotivates the built-in types for numbers and numeric primitives.

[ESOP-2013] Asumu Takikawa, T. Stephen Strickland, and Sam Tobin-Hochstadt, "Constraining Delimited Control with Contracts," European Symposium on Programming, 2013. https://www2.ccs.neu.edu/racket/pubs/esop13-tsth.pdfShows how to type check the % and fcontrol operators in the presence of continuation marks.

[RP:DLS-2014] Michael M. Vitousek, Andrew Kent, Jeremy G. Siek, and Jim Baker, "Design and Evaluation of Gradual Typing for Python," Dynamic Languages Symposium, 2014. https://dl.acm.org/doi/10.1145/2775052.2661101

[ECOOP-2015] Asumu Takikawa, Daniel Feltey, Earl Dean, Robert Bruce Findler, Matthew Flatt, Sam Tobin-Hochstadt, and Matthias Felleisen, "Toward Practical Gradual Typing," European Conference on Object-Oriented Programming, 2015. https://www2.ccs.neu.edu/racket/pubs/ecoop2015-takikawa-et-al.pdfPresents an implementation, experience report, and performance evaluation for gradually-typed first-class classes.

[ESOP-2016] Ambrose Bonnaire–Sergeant, Rowan Davies, and Sam Tobin-Hochstadt, "Practical Optional Types for Clojure," European Symposium on Programming, 2016. https://link.springer.com/chapter/10.1007/978-3-662-49498-1_4

[PLDI-2016] Andrew Kent and Sam Tobin-Hochstadt, "Occurrence Typing Modulo Theories," Conference on Programming Language Design and Implementation, 2016. https://dl.acm.org/citation.cfm?id=2908091Adds linear integer constraints to Typed Racket's compositional occurrence typing.

[Takikawa] Asumu Takikawa, "The Design, Implementation, and Evaluation of a Gradual Type System for Dynamic Class Composition," Ph.D. dissertation, 2016. https://www2.ccs.neu.edu/racket/pubs/dissertation-takikawa.pdf

[RP:POPL-2017] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek, "Big Types in Little Runtime: Open-World Soundness and Collaborative Blame for Gradual Type Systems," Symposium on Principles of Programming Languages, 2017. https://dl.acm.org/doi/abs/10.1145/3009837.3009849

[POPL-2017] Stephen Chang, Alex Knauth, and Emina Torlak, "Symbolic Types for Lenient Symbolic Execution," Symposium on Principles of Programming Languages, 2017. https://www2.ccs.neu.edu/racket/pubs/popl18-ckt.pdfPresents a typed version of Rosette that distinguishes between concrete and symbolic values. The type system supports occurrence typing.

[SNAPL-2017] Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa, "Migratory Typing: Ten Years Later," Summit oN Advances in Programming Languages, 2017. https://www2.ccs.neu.edu/racket/pubs/typed-racket.pdfReflects on origins and successes; looks ahead to current and future challenges.

[KafKa-2018] Benjamin W. Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek, "KafKa: Gradual Typing for Objects," European Conference on Object-Oriented Programming, 2018. https://drops.dagstuhl.de/opus/volltexte/2018/9217/

[Kent-2019] Andrew M. Kent, "Advanced Logical Type Systems for Untyped Languages," Ph.D. dissertation, 2019. https://pnwamk.github.io/docs/dissertation.pdf

[RP:Vitousek-2019] Michael M. Vitousek, "Gradual Typing for Python, Unguarded," Ph.D. dissertation, 2019. https://hdl.handle.net/2022/23172

[OOPSLA-2019] Ben Greenman, Matthias Felleisen, and Christos Dimoulas, "Complete Monitors for Gradual Types," Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2019. https://www2.ccs.neu.edu/racket/pubs/oopsla19-gfd.pdf

[RP:DLS-2019] Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri, "Optimizing and Evaluating Transient Gradual Typing," Dynamic Languages Symposium, 2019. https://dl.acm.org/doi/10.1145/3359619.3359742

[Bonnaire-Sergeant-2019] Ambrose Bonnaire–Sergeant, "Typed Clojure in Theory and Practice," Ph.D. dissertation, 2019. https://scholarworks.iu.edu/dspace/handle/2022/23207

[Greenman-2020] Ben Greenman, "Deep and Shallow Types," Ph.D. dissertation, 2020. http://hdl.handle.net/2047/D20398329

[Programming-2022] Ben Greenman, Lukas Lazarek, Christos Dimoulas, and Matthias Felleisen, "A Transient Semantics for Typed Racket," The Art, Science, and Engineering of Programming 6.2, 2022. https://www2.ccs.neu.edu/racket/pubs/programming-gldf.pdfReports on the difficulties of adapting the Transient semantics of Reticulated Python to the rich migratory type system and established complier infrastructure of Typed Racket.

[PLDI-2022] Ben Greenman, "Deep and Shallow Types for Gradual Languages," Conference on Programming Language Design and Implementation, 2022. http://cs.brown.edu/~bgreenma/publications/apples-to-apples/g-pldi-2022.pdfPresents a language design that combines Deep and Shallow types, and reports on its implementation in Typed Racket.

Index