The Typed Racket Reference

Version 5.1.2

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

August 3, 2011

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

1 Type Reference

`Any`

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

`Nothing`

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

1.1 Base Types

1.1.1 Numeric Types

```
Number
Complex
Float-Complex
Real
Float
Nonnegative-Float
Inexact-Real
Exact-Rational
Integer
Natural
Exact-Nonnegative-Integer
Exact-Positive-Integer
Fixnum
Nonnegative-Fixnum
Positive-Fixnum
Zero
Byte
Exact-Number
Float-Negative-Zero
Float-Positive-Zero
Float-Zero
Flonum
Flonum-Negative-Zero
Flonum-Positive-Zero
Flonum-Zero
Index
```

```
Inexact-Complex
Inexact-Real-Negative-Zero
Inexact-Real-Positive-Zero
Inexact-Real-Zero
Negative-Exact-Rational
Negative-Float
Negative-Flonum
Negative-Inexact-Real
Negative-Integer
Negative-Real
Negative-Single-Flonum
Nonnegative-Exact-Rational
Nonnegative-Flonum
Nonnegative-Inexact-Real
Nonnegative-Real
Nonnegative-Single-Flonum
Nonpositive-Exact-Rational
Nonpositive-Fixnum
Nonpositive-Float
Nonpositive-Flonum
Nonpositive-Inexact-Real
Nonpositive-Integer
Nonpositive-Real
Nonpositive-Single-Flonum
One
Positive-Byte
Positive-Exact-Rational
Positive-Float
Positive-Flonum
Positive-Index
Positive-Inexact-Real
Positive-Integer
Positive-Real
Positive-Single-Flonum
Real-Zero
Single-Flonum
Single-Flonum-Complex
Single-Flonum-Negative-Zero
Single-Flonum-Positive-Zero
Single-Flonum-Zero
```

These types represent the hierarchy of numbers of Racket. `Integer` includes only integers that are exact numbers, corresponding to the predicate `exact-integer?`. `Real` includes both exact and inexact reals. An `Inexact-Real` can be either 32- or 64-bit floating-point numbers. `Float` is restricted to 64-bit floats, which are the default in Racket.

Examples:

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

1.1.2 Other Base Types

```
Boolean
True
False
String
Keyword
Symbol
Char
Void
Input-Port
Output-Port
Port
Path
Path-String
Path-For-Some-System
Regexp
PRegexp
Byte-Regexp
Byte-PRegexp
Bytes
Namespace
Namespace-Anchor
```

```
Variable-Reference
Null
EOF
Continuation-Mark-Set
Prompt-Tag
Undefined
Module-Path
Module-Path-Index
Resolved-Module-Path
Compiled-Module-Expression
Compiled-Expression
Internal-Definition-Context
Pretty-Print-Style-Table
Special-Comment
Struct-Type-Property
Impersonator-Property
Read-Table
Bytes-Converter
Parameterization
Custodian
Inspector
Security-Guard
UDP-Socket
TCP-Listener
Logger
Log-Receiver
Log-Level
Thread
Thread-Group
Subprocess
Place
Place-Channel
Semaphore
Will-Executor
Pseudo-Random-Generator
```

These types represent primitive Racket data.

Examples:

```
> #t
- : True
#t
> #f
- : False
#f
```

```
> "hello"
"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*"
- : Regexp
#px"a*b*"
> '#"bytes"
- : Bytes
#"bytes"
> (current-namespace)
- : Namespace
#<namespace:0>
> #\b
- : Char
#\b
> (thread (lambda () (add1 7)))
- : Thread
#<thread>
```

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 Containers

The following base types are parametric in their type arguments.

`(Pairof s t)`

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

Examples:

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

`(Listof t)`

Homogenous lists of *t*

`(List t ...)`

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

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

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

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

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

Examples:

```
> (list 'a 'b 'c)
- : (List 'a 'b 'c)
```

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

`(MListof t)`

Homogenous mutable lists of *t*.

`(MPairof t u)`

Mutable pairs of *t* and *u*.

`(Boxof t)`

A box of *t*

Example:

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

`(Vectorof t)`

Homogenous vectors of *t*

`(Vector t ...)`

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

`FlVector`

An flvector.

Examples:

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

```
(HashTable k v)
```

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

Example:

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

```
(Setof t)
```

is the type of a set of *t*.

Example:

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

```
(Channelof t)
```

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

Example:

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

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

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

Examples:

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

(Promise *t*)

A promise of *t*.

Example:

```
> (delay 3)
- : (Promise Positive-Byte)
#<promise:eval:35:0>
```

(Futureof *t*)

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

(Sequenceof *t* ...)

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

(Custodian-Boxof *t*)

A custodian box of *t*.

(Thread-Cellof *t*)

A thread cell of *t*.

1.4 Syntax Objects

The following types represent syntax objects and their content.

(Syntaxof *t*)

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

Identifier

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

Syntax

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

`Syntax-E`

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

`(Sexpof t)`

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

`Sexp`

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

`Datum`

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

`(Ephemeronof t)`

An ephemeron whose value is of type *t*.

1.5 Other Type Constructors

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

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

Examples:

```
> (λ: ([x : Number]) x)
- : (Number -> Number : ((! False @ 0) | (False @ 0)) (0))
#<procedure>
> (λ: ([x : Number]    y : String *)  (length y))
- : (Number String * -> Index)
#<procedure>
> ormap
- : (All (a c b ...) ((a b ... b -> c) (Listof a) (Listof b) ... b
-> c))
#<procedure:ormap>
> string?
- : (Any -> Boolean : String)
#<procedure:string?>
```

`Procedure`

is the supertype of all function types.

`(U t ...)`

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

Example:

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

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

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

Example:

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

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

`(t t1 t2 ...)`

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

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

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

Examples:

```
> (: list-lenght : (All (A) (Listof A) -> Natural))
> (define (list-lenght lst)
    (if (null? lst)
        0
        (add1 (list-lenght (cdr lst)))))
```

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

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

Example:

```
> (values 1 2 3)
- : (Values One Positive-Byte Positive-Byte)
1
2
3
```

```
v
```

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

```
(quote val)
```

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

```
i
```

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

```
(Rec n t)
```

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

Examples:

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

→

An alias for `->`.

∀

An alias for `All`.

1.6 Other Types

`(Option t)`

Either *t* or `#f`

`(Opaque t)`

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

2 Special Form Reference

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

2.1 Binding Forms

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

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

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

Examples:

```
> (: filter-even : (Listof Natural) (Listof Natural) -> (Listof Natural))
> (define (filter-even lst accum)
    (if (null? lst)
        accum
        (let: ([first : Natural (car lst)]
               [rest  : (Listof Natural) (cdr lst)])
              (if (even? first)
                  (filter-even rest (cons first accum))
                  (filter-even rest accum)))))
> (filter-even (list 1 2 3 4 5 6) null)
- : (Listof Natural)
'(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 Natural)
'(4 2)
```

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

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

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

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

2.2 Anonymous Functions

```
(lambda: formals . body)

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

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

```
(λ: formals . body)
```

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

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

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

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

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

Example:

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

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

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

A polymorphic function of multiple arities.

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

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

A function with optional arguments.

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

A polymorphic function with optional arguments.

2.3 Loops

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

type-ann-maybe =
               | : u

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

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

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

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

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

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

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

These behave like their non-annotated counterparts.

```
for
for*
```

These are identical to for and for*, but provide additional annotations to help the type-checker.

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

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

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

2.4 Definitions

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

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

Examples:

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

2.5 Structure Definitions

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

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

      name-spec = name
                | name parent

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

Defines a structure with the name *name*, where the fields *f* have types *t*, similar to the behavior of `struct`. When *parent* is present, the structure is a substructure of *parent*. When *maybe-type-vars* is present, the structure is polymorphic in the type variables *v*.

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

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

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

      name-spec = name
                | (name parent)

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

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

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

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

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

2.6 Names for Types

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

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

Examples:

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

2.7 Generating Predicates Automatically

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

Defines `name` as a predicate for the type `t`. `name` has the type `(Any -> Boolean : t)`. `t` may not contain function types.

2.8 Type Annotation and Instantiation

```
(: v t)
```

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

Examples:

```
> (: var1 Integer)
> (: var2 String)
```

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

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

```
(litchar " #{v : t} ")
```

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

```
(ann e t)
```

Ensure that *e* has type *t*, or some subtype. The entire expression has type *t*. This is legal only in expression contexts. The syntax `#{e :: t}` may also be used.

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

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

Example:

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

Examples:

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

The syntax `#{e @ t ...}` may also be used.

2.9 Require

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

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

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

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

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

The first form requires r, giving it type t.

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

Examples:

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

    (struct IntTree
      (elem left right))

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

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

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

Some types, notably polymorphic types constructed with All, 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->.}

2.10 Other Forms

```
with-handlers
```

Identical to with-handlers, but provides additional annotations to help the typechecker.

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

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

```
(#%top-interaction . form)
```

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

3 Libraries Provided With Typed Racket

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

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

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

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

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

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

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

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

4 Utilities

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

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

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

Examples:

```
> (define: x : (U #f String) (number->string 7))
> x
- : (U String False)
"7"
> (assert x)
- : String
"7"
> (define: y : (U String Symbol) "hello")
> y
- : (U Symbol String)
"hello"
> (assert y string?)
- : String
"hello"
> (assert y boolean?)
Assertion failed
```

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

maybe-pred =
           | predicate
```

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

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

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

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

A predicate for the `Index` type.

5 Typed Racket Syntax Without Type Checking

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

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

Examples:

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

6 Typed Regions

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

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

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

result-spec = #:result type

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

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

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

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

Examples:

```
> (with-type #:result Number 3)
3
> ((with-type #:result (Number -> Number)
     (lambda: ([x : Number]) (add1 x)))
   #f)
contract violation: expected <Number>, given: #f
  contract from (region typed-region), blaming top-level
  contract: (-> Number Number)
> (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: self-contract violation, expected <String>, given: 'hello
  contract from top-level, blaming top-level
```

contract: String
at: eval:5.0

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

7 Optimization in Typed Racket

[1]

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

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

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

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

8 Legacy Forms

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

`define-type-alias`

Equivalent to `define-type`.

`define-typed-struct`

Equivalent to `define-struct:`

`require/opaque-type`

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

`require-typed-struct`

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

`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`.

9 Compatibility Languages

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

Typed versions of the

```
#lang scheme
```

and

```
#lang scheme/base
```

languages. The

```
#lang typed-scheme
```

language is equivalent to the

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

language.

10 Experimental Features

These features are currently experimental and subject to change.

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

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

```
(Instance c)
```

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

```
(:type t)
```

Prints the type *t*.

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

Declares *id* to be usable in refinement types.

```
(Refinement id)
```

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

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

Defines an executable structure.