

The Typed Scheme Reference

Version 4.2.3

Sam Tobin-Hochstadt

December 1, 2009

`#lang typed-scheme`

1 Type Reference

Base Types

Number
Integer
Boolean
String
Keyword
Symbol
Void
Input-Port
Output-Port
Path
Regexp
PRegexp
Syntax
Identifier
Bytes
Namespace
EOF
Continuation-Mark-Set
Char

These types represent primitive Scheme data. Note that Integer represents exact integers.

Any

Any Scheme value. All other types are subtypes of Any.

The following base types are parameteric in their type arguments.

(Listof *t*)

Homogenous lists of *t*

(Boxof *t*)

A box of *t*

(Syntaxof *t*)

A syntax object containing a `t`

`(Vectorof t)`

Homogenous vectors of `t`

`(Option t)`

Either `t` of `#f`

`(Parameter t)`

`(Parameter 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.

`(Pair s t)`

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

`(HashTable k v)`

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

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.

`(U t ...)`

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

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

is a function that behaves like all of the *fun-tys*. The *fun-tys* must all be function types constructed with `->`.

`(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 ...*

`(List t ...)`

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

`(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.

v

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

`(quote val)`

where *val* is a Scheme 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*

Other types cannot be written by the programmer, but are used internally and may appear in error messages.

`(struct:n (t ...))`

is the type of structures named `n` with field types `t`. There may be multiple such types with the same printed representation.

`<n>`

is the printed representation of a reference to the type variable `n`

2 Special Form Reference

Typed Scheme provides a variety of special forms above and beyond those in PLT Scheme. 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*).

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

Type-annotated versions of `letrec` and `let*`.

```
(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)
```

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.

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

A polymorphic function of multiple arities.

2.3 Loops

```
(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`.

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`:

2.5 Structure Definitions

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

```
maybe-type-vars =  
    | (v ...)  
  
    name-spec = name  
    | (name parent)
```

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

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

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

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

2.6 Type Aliases

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

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

2.7 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.

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

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

`#{v : t}` This declares that the variable *v* has type *t*. This is legal only for binding occurrences of *v*.

```
(ann e t)
```

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

`#{e :: t}` This is identical to `(ann e t)`.

```
(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.

`#{e @ t ...}` This is identical to `(inst e t ...)`.

2.8 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 (name parent) ([f : t] ...)]
             | [opaque t pred]
```

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 Scheme, either built-in or via `require/typed`. The structure predicate has the appropriate Typed Scheme filter type so that it may be used as a predicate in `if` expressions in Typed Scheme.

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 $(Any \rightarrow Boolean)$. 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.

3 Libraries Provided With Typed Scheme

The `typed-scheme` language corresponds to the `scheme/base` language—that is, any identifier provided by `scheme/base`, such as `mod` is available by default in `typed-scheme`.

```
#lang typed-scheme
(modulo 12 2)
```

Any value provided by `scheme` is available by simply requiring it; use of `require/typed` is not necessary.

```
#lang typed-scheme
(require scheme/list)
(display (first (list 1 2 3)))
```

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-scheme
(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 Scheme via `require/typed`.

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

4 Typed Scheme Syntax Without Type Checking

```
#lang typed-scheme/no-check
```

On occasions where the Typed Scheme syntax is useful, but actual typechecking is not desired, the `typed-scheme/no-check` language is useful. It provides the same bindings and syntax as Typed Scheme, but does no type checking.

Examples:

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