

Package ‘sicher’

October 16, 2026

Type Package

Title Runtime Type Checking

Version 0.2.0

Description Provides a lightweight runtime type system for 'R' that enables developers to declare and enforce variable types during execution. Inspired by 'TypeScript', the package introduces intuitive syntax for annotating variables and validating data structures, helping catch type-related errors early and making 'R' code more robust and easier to maintain.

License MIT + file LICENSE

Encoding UTF-8

URL <https://github.com/feddelegrand7/sicher>

BugReports <https://github.com/feddelegrand7/sicher/issues>

RoxygenNote 7.3.2

Depends R (>= 4.0.0)

Imports glue (>= 1.8.0), methods, utils

Suggests jsonlite, testthat, withr

NeedsCompilation no

Author Mohamed El Fodil Ihaddaden [aut, cre]

Maintainer Mohamed El Fodil Ihaddaden <ihaddaden.fodeil@gmail.com>

Repository CRAN

Date/Publication 2026-10-16 13:20:07 UTC

Contents

Any	3
Between	3
Bool	4
branded-types	4
check_type	5

Coerce	6
create_dataframe_type	7
create_list_type	8
create_type	9
DataFrame	10
Date	11
DateTime	11
Default	12
DiscriminatedUnion	12
Double	13
Enum	14
Environment	14
extend	15
Factor	16
Function	16
FunctionOf	17
infer_type	17
InstanceOf	18
Integer	19
intersection-operator	19
is_type	20
Lazy	21
length-modifiers	22
List	22
ListOf	23
Literal	23
Matches	24
MatrixOf	25
NonEmpty	26
NonNA	26
NonNegative	27
Null	27
Nullable	28
Numeric	28
object-helpers	29
Optional	30
parse_as	31
Positive	32
print.sicher_parse_result	32
print.sicher_readonly	33
print.sicher_type	33
print.sicher_typed_annotation	34
print.sicher_typed_function	34
print.sicher_union	35
Readonly	35
Record	36
Refine	36
Scalar	37

Any

3

String

38

to_json_schema

38

Tuple

39

typed_function

40

type_of

41

union-operator

42

Unique

42

untype

43

validate_json

44

Whole

45

[.sicher_type

45

%:%%

46

%<-%

46

Index

48

Any	<i>Built-in Any Type</i>
-----	--------------------------

Description

A type that accepts any value.

Usage

Any

Format

An object of class `sicher_type` of length 2.

Between	<i>Between — closed-interval numeric range constraint</i>
---------	---

Description

Creates a type that accepts numeric values within the closed interval `[min, max]`. Every element of a vector must satisfy the bounds. NA values are always rejected.

Usage

Between(min, max)

Arguments

- min

A single non-NA numeric scalar giving the lower bound (inclusive).
- max

A single non-NA numeric scalar giving the upper bound (inclusive).

Value

A new `sicher_type` that checks the value is numeric, NA-free, and that all elements lie within `[min, max]`.

Examples

```
age  %:% Between(0, 150)  %<-% 30
score %:% Between(0.0, 1.0) %<-% 0.95
try(score <- 1.5)  # Error: 1.5 is outside [0, 1]
try(score <- NA_real_)  # Error: value contains NA(s)

# Works with integer storage (is.numeric(1L) is TRUE in R)
count %:% Between(0L, 100L) %<-% 42L
```

Bool	<i>Built-in Boolean Type</i>
------	------------------------------

Description

A type that checks for logical vectors.

Usage

```
Bool
```

Format

An object of class `sicher_type` of length 2.

branded-types	<i>Branded types</i>
---------------	----------------------

Description

`Brand()` creates a nominal type on top of a structural one, like `zod's .brand()`. A branded value carries the brand as a class, so an `OrderId` can no longer be passed where a `UserId` is expected even though both are strings. Use `brand()` to check a value against the base type and attach the brand.

Usage

```
Brand(type, brand)

brand(value, type)
```

Arguments

type	A <code>sicher_type</code> or <code>sicher_union</code> object (the base type).
brand	A single string naming the brand. It is added as a class to branded values.
value	A value to brand.

Value

`Brand()` returns a `sicher_type`; `brand()` returns value with the brand class added.

Examples

```
UserId <- Brand(Scalar(String), "UserId")
OrderId <- Brand(Scalar(String), "OrderId")

uid <- brand("u-123", UserId)
current_user %:% UserId %<-% uid

try(current_user <- "u-456")           # Error: not branded
try(current_user <- brand("o-789", OrderId)) # Error: wrong brand
```

check_type	<i>Type Checking Function</i>
------------	-------------------------------

Description

Validates that a value conforms to a specified type. This is the core validation function used internally by the type system, but can also be called directly for manual type checking.

Usage

```
check_type(value, type, context = NULL)
```

Arguments

value	The value to check
type	A <code>sicher_type</code> , <code>sicher_union</code> , or <code>sicher_readonly</code> object
context	Optional character string describing where the check is occurring (used in error messages)

Details

This function:

- Checks if a value matches a type specification
- Handles union types (checks if value matches any type in the union)

- Handles readonly types (strips the readonly modifier before checking)
- Reports every problem it finds in nested lists, data frames and `ListOf()` values, each with its location (e.g. `todos[[2]]$title`)

The condition thrown on failure has class `"sicher_type_error"` and an `issues` element holding the individual problems, so it can be caught with `tryCatch(..., sicher_type_error = function(e) e$issues)`. Use `is_type()` or `safe_parse()` for a check that does not throw.

Value

Returns `'TRUE'` if the value matches the type, otherwise throws an error of class `"sicher_type_error"` with a descriptive message.

See Also

`create_type` for creating custom types, `is_type`, `safe_parse`, `parse_as`

Examples

```
# Direct type checking
check_type(5L, Integer) # Returns TRUE
try(check_type("hello", Integer)) # Throws error

# With context for better error messages
try(check_type(5L, String, context = "user_name"))

# With union types
check_type(5L, Integer | String) # Returns TRUE
try(check_type(5.5, Integer | String)) # Throws error
```

Coerce

Coerce values before checking them

Description

Converts values to the target type before checking them, like `zod's z.coerce`. Values that already satisfy type are left alone. Conversion uses the matching `as.*()` function for built-in types (`as.numeric`, `as.integer`, `as.character`, `as.logical`, `as.Date`, `as.POSIXct`, `factor`, ...), or the function you pass as `using`. A conversion that introduces new NA values (e.g. `as.numeric("abc")`) is reported as an error instead of silently producing NA.

Typed variables, typed function arguments, `parse_as()` and `safe_parse()` store the converted value.

Usage

```
Coerce(type, using = NULL)
```

Arguments

type	A <code>sicher_type</code> . Built-in types and modifiers of built-in types (e.g. <code>Scalar(Numeric)</code> , <code>Between(0, 1)</code>) know their conversion; other types need using.
using	Optional conversion function.

Value

A new `sicher_type`.

Examples

```
age %:% Coerce(Numeric) %<-% "42"
age      # 42 (a number)
try(age <- "forty-two")  # Error: cannot convert

when %:% Coerce(Date) %<-% "2024-05-01"
class(when)  # "Date"
```

`create_dataframe_type` *Create a Data Frame Type with Column Specification*

Description

Builds a type that validates a data frame's column names and their types. Each column is treated as a vector and checked against the provided `sicher_type` (or union) specification. Optional columns may be declared with `Optional()`.

Usage

```
create_dataframe_type(col_spec, extra = c("error", "allow", "drop"))
```

Arguments

col_spec	A named list where names are column names and values are <code>sicher_type</code> or <code>sicher_union</code> objects describing the expected column type.
extra	What to do with columns that are not part of <code>col_spec</code> : "error" (the default), "allow" or "drop". See create_list_type() .

Value

A `sicher_type` representing the data frame schema.

Examples

```

PersonDF <- create_dataframe_type(list(
  name = String,
  age = Numeric,
  height = Optional(Numeric)
))

df %:> PersonDF %<-% data.frame(
  name = c("Alice", "Bob"),
  age = c(25, 30)
)

```

create_list_type	Create a List Type with Specific Structure
------------------	--

Description

Creates a type that checks for lists with specific named elements and their types. Similar to object types in TypeScript/JavaScript.

Usage

```
create_list_type(type_spec, extra = c("error", "allow", "drop"))
```

Arguments

type_spec	A named list where names are field names and values are <code>sicher_type</code> objects
extra	What to do with fields that are not part of <code>type_spec</code> : "error" (the default) rejects them, "allow" keeps them unchecked, and "drop" removes them from the parsed value (similar to zod's <code>.strict()</code> , <code>.passthrough()</code> and <code>.strip()</code>). Typed variables and <code>parse_as()</code> store the parsed value, so with "drop" unknown fields are silently removed.

Value

A `sicher_type` that validates list structure

Examples

```

# Define a User type
User <- create_list_type(list(
  name = String,
  age = Numeric,
  preferences = create_list_type(list(
    color = String,
    movie = String
  ))
))

```

```

))

# Use it
user :: User <- list(
  name = "Alice",
  age = 25,
  preferences = list(color = "red", movie = "batman")
)

```

create_type

Create a Custom Type

Description

Creates a new type object for use in the type checking system. A type consists of a name (for error messages) and a checker function (for validation).

Usage

```
create_type(name, checker)
```

Arguments

name	A single character string representing the type name. This name will be displayed in error messages when type checking fails.
checker	A function that takes a single argument and returns ‘TRUE’ if the value matches the type, ‘FALSE’ otherwise. The checker function should be a predicate (e.g., ‘is.numeric’, ‘is.character’).

Details

This is the fundamental building block of the type system. Built-in types like ‘Integer’, ‘Double’, and ‘String’ are all created using this function.

The checker function should:

- Accept a single argument (the value to check)
- Return ‘TRUE’ if the value is valid for this type
- Return ‘FALSE’ if the value is invalid
- Not throw errors (error handling is done by ‘check_type’)

Value

An object of class “sicher_type” containing:

name The type name as a character string

check The checker function

See Also

[check_type](#) for type validation, [Scalar](#) for creating scalar type variants, [Readonly](#) for creating readonly type variants

Examples

```
# Create a custom positive number
Positive <- create_type("positive", function(x) {
  is.numeric(x) && all(x > 0)
})

# Use it in type annotations
age %:% Positive %<-% 25
try(age <- -5) # Error: Type error

# Create a custom email type
Email <- create_type("email", function(x) {
  is.character(x) &&
  length(x) == 1 &&
  grepl("^[^@]+@[^@]+\\.([^@]+)$", x)
})

user_email %:% Email %<-% "user@example.com"

# Create a type for even integers
EvenInt <- create_type("even_int", function(x) {
  is.integer(x) && all(x %% 2 == 0)
})

value %:% EvenInt %<-% 4L
try(value <- 5L) # Error: Type error

# Create a type that checks data frame structure
PersonDF <- create_type("person_df", function(x) {
  is.data.frame(x) &&
  all(c("name", "age") %in% names(x)) &&
  is.character(x$name) &&
  is.numeric(x$age)
})
```

 DataFrame

Built-in DataFrame Type

Description

A type that checks for data.frame objects.

Usage

DataFrame

Format

An object of class `sicher_type` of length 2.

Date	<i>Built-in Date Type</i>
------	---------------------------

Description

A type that checks for Date vectors.

Usage

Date

Format

An object of class `sicher_type` of length 2.

Examples

```
start %:% Date %<-% as.Date("2024-01-01")
try(start <- "2024-01-01") # Error: a string is not a Date
```

DateTime	<i>Built-in DateTime Type</i>
----------	-------------------------------

Description

A type that checks for date-time (POSIXct) vectors.

Usage

DateTime

Format

An object of class `sicher_type` of length 2.

Examples

```
created %:% DateTime %<-% Sys.time()
```

Default	<i>Default value for a type</i>
---------	---------------------------------

Description

Replaces NULL with a default value, like zod's `.default()`. Inside `create_list_type()` a field with a default may be left out and is filled in with the default value. Any other value is checked against type.

Typed variables, typed function arguments, `parse_as()` and `safe_parse()` all store the parsed value, so the default is what ends up in the variable.

Usage

`Default(type, value)`

Arguments

- `type` A `sicher_type` or `sicher_union` object.
- `value` The default value. It must itself satisfy `type`.

Value

A new `sicher_type`.

Examples

```
Config <- create_list_type(list(
  host = String,
  port = Default(Numeric, 8080)
))
parse_as(list(host = "localhost"), Config)  # port is filled in

retries %:% Default(Integer, 3L) %<-% NULL
retries  # 3L
```

DiscriminatedUnion	<i>Discriminated Union Type Factory</i>
--------------------	---

Description

Creates a union of list types that share a "discriminator" field, like zod's `z.discriminatedUnion()` and TypeScript's tagged unions. The discriminator's value picks which member to check, so errors point at the right fields instead of reporting that nothing in the union matched.

Usage

```
DiscriminatedUnion(key, ...)
```

Arguments

key	Name of the discriminator field.
...	The member types, created with <code>create_list_type()</code> . Either name each argument with its discriminator value (the field is then added to the member automatically if it does not have it), or leave them unnamed and give each member a <code>key = Literal(value)</code> field.

Value

A `sicher_type`.

Examples

```
Circle <- create_list_type(list(radius = Numeric))
Square <- create_list_type(list(side = Numeric))

Shape <- DiscriminatedUnion("kind", circle = Circle, square = Square)

shape %:: Shape %<-% list(kind = "circle", radius = 2)
shape <- list(kind = "square", side = 3)
try(shape <- list(kind = "square", radius = 3)) # Error: points at 'side'
try(shape <- list(kind = "hexagon"))           # Error: unknown kind
```

Double

Built-in Double Type

Description

A type that checks for double-precision numeric vectors.

Usage

```
Double
```

Format

An object of class `sicher_type` of length 2.

Enum

Enum Type Factory

Description

Creates an enumeration type using regular function call syntax. The resulting type only accepts atomic vectors whose elements all belong to the declared set of allowed values.

Usage

```
Enum(...)
```

Arguments

... Allowed scalar values or a single atomic vector of allowed values.

Value

A new `sicher_type` that checks all values belong to the enum.

Examples

```
status %:> Enum(1, 2, 3) %<-% 2
colors %:> Enum("red", "green", "blue") %<-% c("red", "blue")
try(colors <- c("yellow", "red"))
```

Environment

Built-in Environment Type

Description

A type that checks for environments.

Usage

```
Environment
```

Format

An object of class `sicher_type` of length 2.

Examples

```
cache %:> Environment %<-% new.env()
```

extend	<i>Extend a structured list type with additional fields</i>
--------	---

Description

Creates a new list type by merging the field specification of an existing `create_list_type()` type with a set of additional fields. Analogous to TypeScript interface extension (interface `Employee` extends `Person`).

Usage

```
extend(base, extra)
```

Arguments

base	A <code>sicher_type</code> produced by <code>create_list_type()</code> . Must carry a <code>sicher_spec</code> attribute (all types built with the patched <code>create_list_type()</code> above automatically do).
extra	A named list of additional fields in the same format accepted by <code>create_list_type()</code> – names are field names, values are <code>sicher_type</code> or <code>sicher_union</code> objects.

Value

A new `sicher_type` whose required and optional fields are the union of `base`'s fields and `extra`'s fields. Fields in `extra` that share a name with a field in `base` *override* the base field's type (with a warning so the shadowing is never silent).

Examples

```
Person <- create_list_type(list(
  name = String,
  age  = Numeric
))

# Basic extension
Employee <- extend(Person, list(
  role      = String,
  department = Optional(String)
))

emp %:% Employee %<-% list(name = "Alice", age = 30, role = "Engineer")

# Multi-level extension
Manager <- extend(Employee, list(
  reports = Numeric # number of direct reports
))

mgr %:% Manager %<-% list(
  name = "Bob", age = 45, role = "VP", reports = 12
```

```
)  
  
# Field override (emits a warning)  
DetailedPerson <- extend(Person, list(  
  age = Integer    # narrows Numeric -> Integer  
))
```

Factor	<i>Factor Type Factory</i>
--------	----------------------------

Description

Creates a type that checks for factors. When levels is given, the factor’s levels must be exactly levels (same values, same order).

Usage

```
Factor(levels = NULL)
```

Arguments

levels Optional character vector of expected levels.

Value

A `sicher_type`.

Examples

```
size %:% Factor(c("S", "M", "L")) %<-% factor("M", levels = c("S", "M", "L"))  
try(size <- factor("XL"))    # Error: wrong levels  
  
any_factor %:% Factor() %<-% factor(c("a", "b"))
```

Function	<i>Built-in Function Type</i>
----------	-------------------------------

Description

A type that checks for function objects.

Usage

```
Function
```

Format

An object of class `sicher_type` of length 2.

FunctionOf

*Function Type Factory***Description**

Creates a type for functions with a given signature, like TypeScript's `(x: number) => string` and `zod's z.function()`. A value must be a function that accepts the named parameters (or `...`).

Because the types of a function's parameters cannot be known before it is called, the checked function is wrapped with `typed_function()`: each later call checks its arguments and return value. Typed variables, typed function arguments and `parse_as()` store this wrapped function.

Usage

```
FunctionOf(params = list(), .return = NULL)
```

Arguments

<code>params</code>	A named list mapping parameter names to types.
<code>.return</code>	Optional return type.

Value

A `sicher_type`.

Examples

```
Formatter <- FunctionOf(list(x = Numeric), .return = String)

fmt %:>% Formatter %<-% function(x) format(x, nsmall = 2)
fmt(3)                # "3.00"
try(fmt("a"))         # Error: argument x must be numeric

try(fmt <- function(y) y) # Error: no parameter named x
```

infer_type

*Infer a Type from an R Object***Description**

Infers the most appropriate `sicher` type constructor for a given R object. By default, inference focuses on the underlying type and does not lock in the observed length of vectors. Set `'strict = TRUE'` to also infer scalar and fixed-size vector constraints from the example value.

Usage

```
infer_type(obj, strict = FALSE)
```

Arguments

<code>obj</code>	Any R object (primitive, vector, list, data.frame, function, etc.)
<code>strict</code>	Logical scalar. When 'FALSE' (default), infer only the base type shape, such as 'Numeric', 'String', 'ListOf(Integer)', or a 'create_dataframe_type()' schema without fixed lengths. When 'TRUE', also infer 'Scalar()' and '[n]' size constraints from the observed object.

Value

A `sicher_type` object (e.g., `Numeric`, `String`, `create_list_type(...)`, `ListOf(...)`, etc.)

Examples

```
infer_type(42L)           # Integer
infer_type(3.14)          # Double
infer_type(c(1, 2, 3))    # Double or Numeric, no length constraint
infer_type("abc")         # String
infer_type(c("a", "b"))   # String
infer_type(TRUE)          # Bool
infer_type(NULL)          # Null
infer_type(function(x) x + 1) # Function
infer_type(factor(c("a", "b"))) # Factor(c("a", "b"))
infer_type(Sys.Date())     # Date
infer_type(list(a = 1, b = "x")) # create_list_type(list(a = Double, b = String))
infer_type(list(1, 2, 3))   # ListOf(Double)
infer_type(data.frame(x = 1:3)) # create_dataframe_type(list(x = Integer))
infer_type(list(a = NULL, b = 1)) # create_list_type(list(a = Optional(Any), b = Double))

# Strict mode keeps observed length constraints
infer_type(42L, strict = TRUE)      # Scalar(Integer)
infer_type(c("a", "b"), strict = TRUE) # String[2]
```

InstanceOf

Instance-of type

Description

Creates a type that accepts objects inheriting from a class, like `zod's z.instanceof()`. Works with S3 classes (via `inherits()`), R6 classes and S4 classes (via `methods::is()`).

Usage

```
InstanceOf(class)
```

Arguments

<code>class</code>	A character vector of class names. The value must inherit from at least one of them.
--------------------	--

Value

A `sicher_type`.

Examples

```
model %:% InstanceOf("lm") %<-% lm(mpg ~ wt, data = mtcars)
try(model <- list()) # Error: not an lm

when %:% InstanceOf(c("Date", "POSIXct")) %<-% Sys.Date()
```

Integer

Built-in Integer Type

Description

A type that checks for integer vectors.

Usage

Integer

Format

An object of class `sicher_type` of length 2.

intersection-operator *Intersection Type Operator*

Description

S3 methods for the `&` operator to create intersection types, like `A & B` in TypeScript. A value must satisfy both types.

When both sides are structured list types (from `create_list_type()`) the result is a single list type that has the fields of both sides, so `Person & Timestamps` accepts a list with all of the fields. Fields present on both sides must satisfy both field types. When both sides are data frame types the columns are merged the same way. Any other combination produces a type that runs both checks.

Usage

```
## S3 method for class 'sicher_type'
type1 & type2

## S3 method for class 'sicher_union'
type1 & type2
```

Arguments

- type1 First type (sicher_type or sicher_union object)
- type2 Second type (sicher_type or sicher_union object)

Value

A sicher_type

Examples

```
Named <- create_list_type(list(name = String))
Aged  <- create_list_type(list(age = Numeric))

person %:% (Named & Aged) %<-% list(name = "Ann", age = 31)

# Combine a base type with a rule
Score <- Numeric & Between(0, 100)
check_type(42, Score)
```

is_type	<i>Test whether a value matches a type</i>
---------	--

Description

Returns TRUE or FALSE instead of throwing an error. Useful in if conditions, Filter() and vapply().

Usage

```
is_type(value, type)
```

Arguments

- value The value to check.
- type A sicher_type, sicher_union or sicher_readonly object.

Value

A single logical value.

See Also

[check_type](#), [safe_parse](#)

Examples

```
is_type(1, Numeric)      # TRUE
is_type("1", Numeric)    # FALSE

Filter(function(x) is_type(x, String), list("a", 1, "b"))
```

 Lazy

Lazy (recursive) types

Description

Defers building a type until it is first used, like `zod's z.lazy()`. This lets a type refer to itself, which is needed for trees and other recursive structures.

Usage

```
Lazy(fn, name = "lazy")
```

Arguments

<code>fn</code>	A function with no arguments that returns a <code>sicher</code> type. It is called once, the first time a value is checked.
<code>name</code>	Name used when printing the type. Defaults to <code>"lazy"</code> .

Value

A `sicher_type`.

Examples

```
Tree <- create_list_type(list(
  value    = Numeric,
  children = ListOf(Lazy(function() Tree, name = "Tree"))
))

tree %:% Tree %<-% list(
  value = 1,
  children = list(
    list(value = 2, children = list()),
    list(value = 3, children = list())
  )
)
```

length-modifiers	<i>MinLength / MaxLength modifiers</i>
------------------	--

Description

Require a minimum or maximum number of elements. The length is `length()` for vectors and lists and `nrow()` for data frames, consistent with `NonEmpty()` and the `[n]` size operator. To limit the number of characters in a string, use `Matches()` or `Refine()` with `nchar()`.

Usage

```
MinLength(type, n)

MaxLength(type, n)
```

Arguments

- `type` A `sicher_type` or `sicher_union` object.
- `n` A single non-negative whole number.

Value

A new `sicher_type`.

Examples

```
tags %:% MinLength(String, 1) %<-% "r"
try(tags <- character(0)) # Error: fewer than 1 element

top3 %:% MaxLength(Numeric, 3) %<-% c(9, 8, 7)
try(top3 <- c(9, 8, 7, 6)) # Error: more than 3 elements
```

List	<i>Built-in List Type</i>
------	---------------------------

Description

A type that checks for list objects.

Usage

```
List
```

Format

An object of class `sicher_type` of length 2.

ListOf	Create a homogeneous list type
--------	--------------------------------

Description

Produces a type that validates a list whose every element satisfies the provided element type. This is useful when you expect a list of similar records (e.g. parsed JSON array). You can further constrain the length with the size operator: `'ListOf(User)[10]'`.

Usage

```
ListOf(element_type)
```

Arguments

`element_type` A `sicher_type` or `sicher_union` describing each element.

Value

A `sicher_type` that checks the value is a list and that all elements conform to `'element_type'`.

Examples

```
# Define an inner record type
Record <- create_list_type(list(id = Numeric, name = String))

# Now require a list of records
Records <- ListOf(Record)
records %:% Records %<-% list(
  list(id = 1, name = "a"),
  list(id = 2, name = "b")
)

# fixed-size list of ten records
TenRecs <- Records[10]
# will throw if length != 10
```

Literal	Literal Type Factory
---------	----------------------

Description

Creates a literal type inspired by TypeScript literal unions. The resulting type only accepts scalar atomic values that exactly match one of the declared literals, including the underlying R storage mode. For example, `'200'` and `'200L'` are treated as different literals.

Usage

```
Literal(...)
```

Arguments

... Allowed scalar atomic literal values.

Value

A new `sicher_type` that checks the value is exactly one of the declared literals.

Examples

```
direction %:% Literal("left", "right") %<-% "left"
direction <- "right"
direction <- "left"
try(direction <- c("right", "left"))
status_code %:% Literal(200, 404) %<-% 200
try(status_code <- 500)
```

Matches

Matches — regex-constrained string type

Description

Creates a type that accepts only character vectors whose every element matches the given Perl-compatible regular expression. NA elements are always rejected. An empty character vector `character(0)` passes vacuously; combine with `NonEmpty()` if you need at least one element.

Usage

```
Matches(pattern)
```

Arguments

pattern A single non-NA character string used as a PCRE regex (passed to `grep1(..., perl = TRUE)`).

Value

A new `sicher_type` that validates every element against `pattern`.

Examples

```
email %:% Matches("^^[^@]+@[^@]+\\.^[^@]+$") %<-% "user@example.com"
try(email <- "not-an-email") # Error: does not match pattern

hex %:% Matches("^#[0-9A-Fa-f]{6}$") %<-% "#FF5733"

# Combine with NonEmpty to also require at least one element
tags %:% NonEmpty(Matches("^[a-z]+$")) %<-% c("foo", "bar")
```

MatrixOf	<i>Matrix Type Factory</i>
----------	----------------------------

Description

Creates a type that checks for matrices, optionally with a given element type and dimensions. Named MatrixOf (like ListOf) so it does not clash with the Matrix package.

Usage

```
MatrixOf(type = Any, nrow = NULL, ncol = NULL)
```

Arguments

- type Element type, checked against the matrix's values. Defaults to Any.
- nrow, ncol Optional required number of rows and columns.

Value

A sicher_type.

Examples

```
m %:% MatrixOf(Numeric, nrow = 2, ncol = 2) %<-% diag(2)
try(m <- matrix(1:6, nrow = 2)) # Error: wrong dimensions
```

NonEmpty

NonEmpty modifier — require a non-empty value

Description

Creates a type that first validates the underlying type, then rejects empty values. For data frames "empty" means zero rows (`nrow == 0`); for all other objects it means `length == 0`.

Usage

```
NonEmpty(type)
```

Arguments

`type` A `sicher_type` or `sicher_union` object.

Value

A new `sicher_type` that rejects zero-length (or zero-row) values after the base type check passes.

Examples

```
tags %:>% NonEmpty(String) %<-% c("r", "types")
try(tags <- character(0))    # Error: value must be non-empty

items %:>% NonEmpty(List) %<-% list(1, 2)
try(items <- list())        # Error: value must be non-empty
```

NonNA

NonNA modifier — reject values containing NAs

Description

Creates a type that accepts only non-NA values of the underlying type. Any element equal to NA causes an immediate error, making silent NA propagation impossible in typed pipelines.

Usage

```
NonNA(type)
```

Arguments

`type` A `sicher_type` or `sicher_union` object.

Value

A new `sicher_type` that first validates the underlying type, then rejects any value that contains at least one NA.

Examples

```
salary %:> NonNA(Numeric) %<-% c(1800, 2300, 4000)
try(salary <- c(1800, NA, 4000)) # Error: value contains NA(s)

# Composable with other modifiers
tag %:> NonNA(Scalar(String)) %<-% "admin"
```

NonNegative

Built-in NonNegative Type

Description

A numeric type whose elements are all greater than or equal to zero. NA values are rejected. Similar to `zod's z.number().nonnegative()`.

Usage

NonNegative

Format

An object of class `sicher_type` of length 3.

Examples

```
count %:> NonNegative %<-% 0
try(count <- -1) # Error: -1 is not >= 0
```

Null

Built-in Null Type

Description

A type that checks for NULL values.

Usage

Null

Format

An object of class `sicher_type` of length 2.

Nullable	Create a nullable type variant
----------	--------------------------------

Description

Creates a type that accepts NULL in addition to the base type, like Optional(). The difference shows up inside `create_list_type()` and `create_dataframe_type()`: an Optional() field may be left out entirely, while a Nullable() field must be present but may hold NULL. This mirrors the distinction between `field?: T` and `field: T | null` in TypeScript (zod's `.optional()` versus `.nullable()`).

Usage

Nullable(type)

Arguments

type A `sicher_type` or `sicher_union` object.

Value

A union type that includes Null and is marked as nullable.

Examples

```
Profile <- create_list_type(list(
  name      = String,
  nickname  = Optional(String), # may be missing
  bio       = Nullable(String)  # must be present, may be NULL
))

check_type(list(name = "Ann", bio = NULL), Profile)
try(check_type(list(name = "Ann"), Profile)) # Error: 'bio' is missing
```

Numeric	Built-in Numeric Type
---------	-----------------------

Description

A type that checks for numeric vectors (integer or double).

Usage

Numeric

Format

An object of class `sicher_type` of length 2.

object-helpers

Object type utilities: Pick, Omit, Partial, Required, KeyOf

Description

Derive new list (or data frame) types from existing ones, like TypeScript's utility types and the matching zod methods:

- `Pick(type, ...)` keeps only the named fields (`Pick<T, K>`, `zod .pick()`).
- `Omit(type, ...)` removes the named fields (`Omit<T, K>`, `zod .omit()`).
- `Partial(type)` makes every field optional (`Partial<T>`, `zod .partial()`).
- `Required(type)` makes every field required, removing `Optional()` and `Default()` wrappers (`Required<T>`, `zod .required()`). `Nullable()` fields stay nullable.
- `KeyOf(type)` returns an `Enum()` of the field names (`keyof T`, `zod .keyof()`).

The derived types keep the extra setting of the original type.

Usage

```
Pick(type, ...)
```

```
Omit(type, ...)
```

```
Partial(type)
```

```
Required(type)
```

```
KeyOf(type)
```

Arguments

<code>type</code>	A type created with <code>create_list_type()</code> or <code>create_dataframe_type()</code> .
<code>...</code>	Field names, as separate strings or character vectors.

Value

A new `sicher_type`.

Examples

```

User <- create_list_type(list(
  id   = Integer,
  name = String,
  email = Optional(String)
))

UserPreview <- Pick(User, "id", "name")
NewUser     <- Omit(User, "id")
UserPatch   <- Partial(Omit(User, "id"))
FullUser    <- Required(User)
UserField   <- KeyOf(User)

patch %:> UserPatch %<-> list(email = "new@example.com")
field %:> UserField %<-> "email"

```

Optional	<i>Create an optional (nullable) type variant</i>
----------	---

Description

Creates a type that accepts NULL values in addition to the base type.

Usage

```
Optional(type)
```

Arguments

type	A <code>sicher_type</code> object
------	-----------------------------------

Value

A union type that includes Null

Examples

```

middle_name %:> Optional(String) %<-> NULL
middle_name <- "Marie" # Also OK

```

parse_as	<i>Parse a value against a type</i>
----------	-------------------------------------

Description

`parse_as()` checks value against type and returns the parsed value, or throws an error of class "sicher_type_error" that lists every problem found (like `zod's .parse()`).

`safe_parse()` never throws on invalid data (like `zod's .safeParse()`). It returns a result object with:

success TRUE if the value matches the type.

data The parsed value, or NULL on failure.

issues A data frame with one row per problem and the columns path, message, expected, got and code.

error The "sicher_type_error" condition on failure, or NULL.

The parsed value is usually the input value itself, but it can differ when the type transforms values: [Default\(\)](#) fills in missing values, [Coerce\(\)](#) converts them, list types with `extra = "drop"` remove unknown fields and [FunctionOf\(\)](#) wraps functions.

Usage

```
parse_as(value, type, context = NULL)
```

```
safe_parse(value, type, context = NULL)
```

Arguments

value	The value to parse.
type	A <code>sicher_type</code> , <code>sicher_union</code> or <code>sicher_readonly</code> object.
context	Optional name for the value, used in error messages.

Value

`parse_as()` returns the parsed value. `safe_parse()` returns a list of class "sicher_parse_result".

See Also

[check_type](#), [is_type](#), [validate_json](#)

Examples

```

  Todo <- create_list_type(list(
    title = String,
    done  = Default(Bool, FALSE)
  ))

  parse_as(list(title = "Write docs"), Todo)

  result <- safe_parse(list(title = 1, done = "no"), Todo)
  result$success
  result$issues

```

Positive	<i>Built-in Positive Type</i>
----------	-------------------------------

Description

A numeric type whose elements are all strictly greater than zero. NA values are rejected. Similar to `zod's z.number().positive()`.

Usage

```

  Positive

```

Format

An object of class `sicher_type` of length 3.

Examples

```

price %:: Positive %<-% 9.99
try(price <- 0) # Error: 0 is not > 0

```

<code>print.sicher_parse_result</code>	<i>Print method for sicher_parse_result</i>
--	---

Description

Print method for `sicher_parse_result`

Usage

```

## S3 method for class 'sicher_parse_result'
print(x, ...)

```

Arguments

x	A <code>sicher_parse_result</code> object
...	Additional arguments (ignored)

Value

Invisibly returns the input object

`print.sicher_readonly` *Print method for `sicher_readonly`*

Description

Print method for `sicher_readonly`

Usage

```
## S3 method for class 'sicher_readonly'  
print(x, ...)
```

Arguments

x	A <code>sicher_readonly</code> object
...	Additional arguments (ignored)

Value

Invisibly returns the input object

`print.sicher_type` *Print method for `sicher_type`*

Description

Print method for `sicher_type`

Usage

```
## S3 method for class 'sicher_type'  
print(x, ...)
```

Arguments

x	A <code>sicher_type</code> object
...	Additional arguments (ignored)

Value

Invisibly returns the input object

```
print.sicher_typed_annotation
```

Print method for sicher_typed_annotation

Description

Print method for sicher_typed_annotation

Usage

```
## S3 method for class 'sicher_typed_annotation'  
print(x, ...)
```

Arguments

x	A sicher_typed_annotation object
...	Additional arguments (ignored)

Value

Invisibly returns the input object

```
print.sicher_typed_function
```

Print method for sicher_typed_function

Description

Print method for sicher_typed_function

Usage

```
## S3 method for class 'sicher_typed_function'  
print(x, ...)
```

Arguments

x	A sicher_typed_function object
...	Additional arguments (ignored)

Value

Invisibly returns the input object

print.sicher_union	<i>Print method for sicher_union</i>
--------------------	--------------------------------------

Description

Print method for sicher_union

Usage

```
## S3 method for class 'sicher_union'  
print(x, ...)
```

Arguments

x	A sicher_union object
...	Additional arguments (ignored)

Value

Invisibly returns the input object

Readonly	<i>Create a readonly type variant</i>
----------	---------------------------------------

Description

Creates a type that prevents reassignment after initial value is set.

Usage

```
Readonly(type)
```

Arguments

type	A sicher_type object
------	----------------------

Value

A readonly type modifier

Examples

```
PI %:>% Readonly(Double) %<-% 3.14159  
try(PI <- 3.0) # Error: cannot reassign readonly
```

Record

*Record Type Factory***Description**

Creates a type for named collections with arbitrary keys, like TypeScript's `Record<K, V>` and zod's `z.record()`. Every element must be named; each name is checked against keys and each value against values. Both named lists and named atomic vectors are accepted.

Usage

```
Record(keys, values)
```

Arguments

keys	Type of the names. Defaults to <code>String</code> when only one argument is given. Useful choices are <code>Enum()</code> , <code>Literal()</code> and <code>Matches()</code> .
values	Type of each element.

Value

A `sicher_type`.

Examples

```
scores %: Record(String, Numeric) %<- list(alice = 90, bob = 85)

# One argument: keys default to String
prices %: Record(Numeric) %<- c(apple = 1.2, pear = 0.8)

Flags <- Record(Enum("debug", "verbose"), Bool)
try(check_type(list(debug = TRUE, color = FALSE), Flags)) # Error: bad key
```

Refine

*Refine a type with a custom rule***Description**

Adds an extra rule on top of an existing type, like zod's `.refine()`. The base type is checked first; the predicate only runs on values that already have the right type, so it does not need to guard against them.

Usage

```
Refine(type, predicate, message = NULL, name = NULL)
```

Arguments

type	A <code>sicher_type</code> or <code>sicher_union</code> object.
predicate	A function taking the value and returning TRUE if it is valid.
message	Error message used when the predicate does not return TRUE. Defaults to a generic message.
name	Optional type name used when printing the type and in error messages. Defaults to <code>refine<base type></code> .

Value

A new `sicher_type`.

Examples

```

Password <- Refine(
  Scalar(String),
  function(x) nchar(x) >= 8,
  message = "must be at least 8 characters"
)
pw %:% Password %<-% "correct horse"
try(pw <- "short")

Even <- Refine(Integer, function(x) all(x %% 2L == 0L), "must be even", name = "even")
check_type(c(2L, 4L), Even)

```

Scalar	<i>Create a scalar (length-1) type variant</i>
--------	--

Description

Creates a type that only accepts single values (vectors of length 1).

Usage

```
Scalar(type)
```

Arguments

type	A <code>sicher_type</code> object
------	-----------------------------------

Value

A new `sicher_type` that checks for length 1

Examples

```
age %:: Scalar(Integer) %<-% 30L
try(age <- c(30L, 40L)) # Error: not scalar
```

String	<i>Built-in String Type</i>
--------	-----------------------------

Description

A type that checks for character vectors.

Usage

```
String
```

Format

An object of class `sicher_type` of length 2.

to_json_schema	<i>Convert a type to JSON Schema</i>
----------------	--------------------------------------

Description

Converts a `sicher` type to a JSON Schema (draft 2020-12) description, like `zod`'s `z.toJSONSchema()`. Useful for documenting APIs (e.g. `plumber` endpoints) or sharing a schema with other languages.

Because R has no separate scalar type, an unconstrained type such as `Numeric` becomes "a number or an array of numbers"; wrap it in `Scalar()` to get a plain number. Rules that JSON Schema cannot express (`Refine()`, `Coerce()`, `Brand()`, `NonNA()`) are dropped and only their base type is described. Types with no JSON equivalent (`Function`, `Environment`, `InstanceOf()`, `MatrixOf()`, `FunctionOf()`, `Lazy()` and custom `create_type()` types) raise an error.

Usage

```
to_json_schema(type, json = FALSE)
```

Arguments

<code>type</code>	A <code>sicher</code> type.
<code>json</code>	If <code>TRUE</code> , return a JSON string (requires the <code>jsonlite</code> package). If <code>FALSE</code> (the default), return the schema as a nested R list.

Value

A nested list, or a JSON string when `json = TRUE`.

Examples

```
User <- create_list_type(list(
  id   = Scalar(Integer),
  name = Scalar(String),
  email = Optional(Scalar(String))
))
str(to_json_schema(User))

cat(to_json_schema(User, json = TRUE))
```

Tuple	<i>Tuple Type Factory</i>
-------	---------------------------

Description

Creates a type for lists with a fixed number of elements, each with its own type, like TypeScript tuples (`[string, number]`) and zod's `z.tuple()`.

Usage

```
Tuple(...)
```

Arguments

<code>...</code>	The element types, in order. They may be named, which only affects the printed type name.
------------------	---

Value

A `sicher_type`.

Examples

```
Point <- Tuple(Numeric, Numeric)
p %:% Point %<-% list(1.5, 2)

Entry <- Tuple(key = String, value = Numeric)
try(check_type(list("a", "b"), Entry)) # Error at [[2]]
```

typed_function	Create a type-checked function
----------------	--------------------------------

Description

Wraps a function with runtime type checking for its parameters and, optionally, its return value. This is the function counterpart to the typed variable operators ('%:%' / '%<-%'), providing a syntax analogous to typed function signatures:

```
add <- typed_function(
  function(x, y) x + y,
  params = list(x = Numeric, y = Numeric),
  .return = Numeric
)
```

Usage

```
typed_function(fn, params = list(), .return = NULL)
```

Arguments

fn	The function to wrap. Its formals are preserved in the wrapper so callers use the exact same signature.
params	A named list mapping parameter names to their types (e.g. list(x = Numeric, y = String)). Only listed parameters are type-checked on each call; unlisted parameters pass through unchecked. Defaults to an empty list (no parameter checking).
.return	Optional return type. When NULL (the default), the return value is not checked. Accepts any <code>sicher_type</code> or <code>sicher_union</code> .

Value

A function with the same formals as `fn` and S3 class "sicher_typed_function" that:

- Validates each listed parameter on every call.
- Validates the return value when `.return` is specified.
- Delegates all argument passing to `fn` unchanged.

Examples

```
# Basic typed function
add <- typed_function(
  function(x, y) x + y,
  params = list(x = Numeric, y = Numeric),
  .return = Numeric
)
add(1, 2)      # Returns 3
```

```
try(add("a", 2)) # Error: Type error in 'x': Expected numeric, got string

# Optional parameter
greet <- typed_function(
  function(name, title = NULL) {
    if (is.null(title)) paste("Hello,", name)
    else paste("Hello,", title, name)
  },
  params = list(name = String, title = Optional(String))
)
greet("Alice") # "Hello, Alice"
greet("Alice", title = "Dr.") # "Hello, Dr. Alice"
try(greet("Alice", title = 42)) # Error: Type error in 'title'

# Union type in params
describe <- typed_function(
  function(id) paste("ID:", id),
  params = list(id = String | Numeric),
  .return = String
)
describe("abc") # "ID: abc"
describe(123) # "ID: 123"
try(describe(TRUE)) # Error: Type error in 'id'
```

type_of*Get the declared type of a typed variable*

Description

Returns the type a variable was declared with using %:%, similar to TypeScript's typeof in type positions. Returns NULL for variables that are not typed. To guess a type from a value instead, use [infer_type\(\)](#).

Usage

```
type_of(x)
```

Arguments

x A variable name (unquoted).

Value

The declared type (wrapped in Readonly() for readonly variables), or NULL.

Examples

```
age %:> Numeric %<-> 30
type_of(age)          # <type: numeric >

plain <- 1
type_of(plain)        # NULL

# Reuse a variable's type for another one
other_age %:> type_of(age) %<-> 25
```

union-operator	<i>Union Type Operator</i>
----------------	----------------------------

Description

S3 methods for the 'l' operator to create union types.

Usage

```
## S3 method for class 'sicher_type'
type1 | type2

## S3 method for class 'sicher_union'
type1 | type2
```

Arguments

- type1 First type (sicher_type or sicher_union object)
- type2 Second type (sicher_type or sicher_union object)

Value

A union type (sicher_union object)

Unique	<i>Unique modifier</i>
--------	------------------------

Description

Rejects values that contain duplicated elements (for data frames, duplicated rows). Uses anyDuplicated().

Usage

```
Unique(type)
```

Arguments

type A `sicher_type` or `sicher_union` object.

Value

A new `sicher_type`.

Examples

```
ids %:>% Unique(Integer) %<-% c(1L, 2L, 3L)
try(ids <- c(1L, 2L, 2L)) # Error: duplicated value 2
```

untype	<i>Remove the type from a typed variable</i>
--------	--

Description

Turns a typed variable back into a regular variable holding the same value. Later assignments are no longer checked. Does nothing for variables that are not typed.

Usage

```
untype(x)
```

Arguments

x A variable name (unquoted).

Value

The variable's value, invisibly.

Examples

```
count %:>% Integer %<-% 1L
untype(count)
count <- "no longer checked"
```

validate_json

*Parse and validate JSON***Description**

Parses a JSON string with `jsonlite::fromJSON()` and checks the result against a type, returning the parsed value. Handy for checking request bodies in plumber APIs or configuration files.

By default JSON arrays of scalars become atomic vectors (so they match `Numeric`, `String`, ...), JSON objects become named lists (so they match `create_list_type()` types) and arrays of objects become lists of lists (so they match `ListOf()`). Two `jsonlite` behaviours are worth knowing: whole numbers such as 1 are read as integers (use `Numeric` rather than `Double` for JSON numbers), and an empty array `[]` is read as `list()` rather than an empty atomic vector.

Usage

```
validate_json(json, type, safe = FALSE, ...)
```

Arguments

<code>json</code>	A JSON string, URL or file path (anything <code>jsonlite::fromJSON()</code> accepts).
<code>type</code>	A <code>sicher</code> type.
<code>safe</code>	If <code>TRUE</code> , return a safe_parse() result instead of throwing on invalid data.
<code>...</code>	Further arguments passed to <code>jsonlite::fromJSON()</code> , which override the defaults <code>simplifyVector = TRUE</code> , <code>simplifyDataFrame = FALSE</code> and <code>simplifyMatrix = FALSE</code> .

Value

The parsed value, or a `"sicher_parse_result"` when `safe = TRUE`.

Examples

```
Order <- create_list_type(list(
  id    = Scalar(Integer),
  items = ListOf(create_list_type(list(sku = String, qty = Whole)))
))

validate_json('{"id": 1, "items": [{"sku": "A1", "qty": 2}]}', Order)

res <- validate_json('{"id": "x", "items": []}', Order, safe = TRUE)
res$issues
```

Whole	<i>Built-in Whole Type</i>
-------	----------------------------

Description

A numeric type whose elements are all finite whole numbers. Unlike Integer, the storage mode does not matter: both 3L and 3 are accepted, but 3.5 is not. Similar to zod's `z.number().int()`.

Usage

Whole

Format

An object of class `sicher_type` of length 3.

Examples

```
n :% Whole %<-% 3      # a double, but a whole number
n <- 10L                # integers are fine too
try(n <- 2.5)           # Error: not a whole number
```

<code>[.sicher_type</code>	<i>Vector Size Operator for <code>sicher_type</code></i>
----------------------------	--

Description

Creates a type that checks for a specific vector length.

Usage

```
## S3 method for class 'sicher_type'
type[size]
```

Arguments

- `type` A `sicher_type` object
- `size` The required length (non-negative integer)

Value

A new `sicher_type` that checks for the specified length

Examples

```
vec %: Numeric[3] %<- c(1, 2, 3)
try(vec <- c(1, 2)) # Error: wrong length
try(vec <- c("a", "b", "c")) # Error: wrong type
```

%: %	Type annotation operator
------	--------------------------

Description

Creates a typed variable annotation used together with ‘%<-%’.

Usage

```
name %: % type
```

Arguments

name	Variable name (unevaluated).
type	Type specification (e.g., Integer, String, Double).

Value

A typed annotation object.

Examples

```
x %: Integer %<- 5L
name %: String %<- "Alice"
id %: (Integer | String) %<- 42L
```

%<- %	Type-checked assignment operator
-------	----------------------------------

Description

Completes the typed assignment started with ‘%: %’.

Usage

```
typed_annotation %<- value
```

`%<-%`

47

Arguments

<code>typed_annotation</code>	Result of <code>'%: %'</code> .
<code>value</code>	Value to assign.

Value

Invisibly returns the assigned value.

Examples

```
x %: % Integer %<-% 5L
y %: % Double %<-% 3.14
name %: % String %<-% "Bob"
flag %: % Bool %<-% TRUE
```

Index

* datasets

- Any, [3](#)
- Bool, [4](#)
- DataFrame, [10](#)
- Date, [11](#)
- DateTime, [11](#)
- Double, [13](#)
- Environment, [14](#)
- Function, [16](#)
- Integer, [19](#)
- List, [22](#)
- NonNegative, [27](#)
- Null, [27](#)
- Numeric, [28](#)
- Positive, [32](#)
- String, [38](#)
- Whole, [45](#)
- [.sicher_type, [45](#)
- &.sicher_type (intersection-operator), [19](#)
- &.sicher_union (intersection-operator), [19](#)
- %, [46](#)
- %<-, [46](#)
- Any, [3](#)
- Between, [3](#)
- Bool, [4](#)
- Brand (branded-types), [4](#)
- brand (branded-types), [4](#)
- branded-types, [4](#)
- check_type, [5](#), [10](#), [20](#), [31](#)
- Coerce, [6](#), [31](#)
- create_dataframe_type, [7](#), [28](#), [29](#)
- create_list_type, [7](#), [8](#), [12](#), [13](#), [19](#), [28](#), [29](#)
- create_type, [6](#), [9](#)
- DataFrame, [10](#)

- Date, [11](#)
- DateTime, [11](#)
- Default, [12](#), [31](#)
- DiscriminatedUnion, [12](#)
- Double, [13](#)
- Enum, [14](#)
- Environment, [14](#)
- extend, [15](#)
- Factor, [16](#)
- Function, [16](#)
- FunctionOf, [17](#), [31](#)
- infer_type, [17](#), [41](#)
- InstanceOf, [18](#)
- Integer, [19](#)
- intersection-operator, [19](#)
- is_type, [6](#), [20](#), [31](#)
- KeyOf (object-helpers), [29](#)
- Lazy, [21](#)
- length-modifiers, [22](#)
- List, [22](#)
- ListOf, [23](#)
- Literal, [23](#)
- Matches, [22](#), [24](#)
- MatrixOf, [25](#)
- MaxLength (length-modifiers), [22](#)
- MinLength (length-modifiers), [22](#)
- NonEmpty, [26](#)
- NonNA, [26](#)
- NonNegative, [27](#)
- Null, [27](#)
- Nullable, [28](#)
- Numeric, [28](#)
- object-helpers, [29](#)

Omit (object-helpers), 29
Optional, 30

parse_as, 6, 8, 12, 17, 31
Partial (object-helpers), 29
Pick (object-helpers), 29
Positive, 32
print.sicher_parse_result, 32
print.sicher_readonly, 33
print.sicher_type, 33
print.sicher_typed_annotation, 34
print.sicher_typed_function, 34
print.sicher_union, 35

Readonly, 10, 35
Record, 36
Refine, 22, 36
Required (object-helpers), 29

safe_parse, 6, 12, 20, 44
safe_parse (parse_as), 31
Scalar, 10, 37
String, 38

to_json_schema, 38
Tuple, 39
type_of, 41
typed_function, 17, 40

union-operator, 42
Unique, 42
untype, 43

validate_json, 31, 44

Whole, 45