Valkyrie

Type safe type safety in typst

https://github.com/JamesxX/valkyrie

James R. Swift **tinger <me@tinger.dev>**

Abstract

This package implements type validation, and is targetted mainly at package and template developers. The desired outcome is that it becomes easier for the programmer to quickly put a package together without spending a long time on type safety, but also to make the usage of those packages by end-users less painful by generating useful error messages.

I Example usage

```
#import "@preview/valkyrie:0.1.0" as z

#let my-schema = z.dictionary(
    should-be-string: z.string(),
    complicated-tuple: z.tuple(
        z.email(),
        z.ip(),
        z.either(
            z.string(),
            z.number()
        )
    )
)

#z.parse(
    (
        should-be-string: "This doesn't error",
        complicated-tuple: (
            "neither@does-this.com",
            "NOT AN IP", // Error: Schema validation failed on argument.complicated-
tuple.1:
                         //        String must be a valid IP address
            1
        )
    ),
    my-schema
)
```

II Documentation

This documentation is generated using the Tidy package, and therefore, while every effort is made to ensure it is representative of the API, there may still be errors due to oversight. If you come across such an error, please make an issue (or pull request) on the GitHub repository.

Terminology

As this package introduces several type-like objects, the Tidy style has had these added for clarity. At present, these are `schema` (to represent type-validating objects), `ctx` (to represent the current state of the parsing heuristic), and `scope` (an array of strings that represents the parent object of values being parsed). `internal` represents arguments that, while settable by the end-user, should be reserved for internal or advanced usage.

Generally, users of this package will only need to be aware of the `schema` type.

Specific Language

WILL Indicates a guarantee.

MAY Indicates a possibility without guarantee.

MUST Indicates a requirement. The programmed **MAY** be **ILL-FORMED** if it does not follow the requirement.

SHOULD Indicates a soft requirement, but the program remains correctly-formed if the requirement is not met.

UNDEFINED BEHAVIOUR Behaviour resulting from incorrect API usage. Typically, this behaviour has been forseen by the author, but no effort has been made to handle these edge-cases due to complexity or infrequency.

ILL-FORMED Description of a program using the API in an incorrect way such that it is either outside the scope of the package and/or produces **UNDEFINED BEHAVIOUR**.

Contents

Functions

parse

This is the main function for validating an object against a schema. **WILL** return the given object after validation if successful, or none and **MAY** throw a failed assertion error.

Parameters

```
parse(
  object: any,
  schema: schema,
  ctx: ctx,
  scope: scope
) -> any none
```

object `any`

Object to validate against provided schema. Object **SHOULD** statisfy the schema requirements. An error **MAY** be produced if not.

schema `schema`

Schema against which `object` is validated. **MUST** be a valid valkyrie schema type.

ctx `ctx`

ctx passed to schema validator function, containing flags that **MAY** alter behaviour.

Default: `z-ctx()`

scope `scope`

An array of strings used to generate the string representing the location of a failed requirement within `object`. **MUST** be an array of strings of length greater than or equal to `1`.

Default: `("argument",)`

- z-ctx()

z-ctx

Appends options to a context. Used for setting the context of child parses.

Parameters

```
z-ctx(
  parent: ctx none,
  ..args: arguments
)
```

parent `ctx` or `none`

Current context (if present), to which contextual flags passed in variadic arguments are appended.

Default: `(:)`

..args `arguments`

Variadic contextual flags to set. Positionala rguments are discarded.

Types

- any()

any

Validation schema representing all types. **SHOULD** never produce an error.

Parameters

```
any(
  name: internal,
  default: any none,
  custom: function,
  custom-error: string,
  transform: function
) -> schema
```

default `any` or `none`

Default value to validate is none is provided.

Default: `none`

custom `function`

Function that maps an input to an output. If the function returns `none`, then an error **WILL** be generated using `custom-error`.

Default: `none`

custom-error `string`

Error to return if custom function returns none.

Default: `auto`

transform `function`

Function that maps an input to an output, called after validation.

Default: `it => it`

- array()

array

Valkyrie schema generator for array types. Array entries are validated by a single schema. For arrays with positional requirements, see `tuple()`.

Parameters

```
array(
  name: internal,
  default: array none,
  min: integer none,
  max: integer none,
  length: integer auto,
  custom: function none,
  custom-error: string none,
  transform: function,
  ..args: schema none
) -> schema
```

default `array` or `none`

Default value to set if no value is provided. **MUST** itself pass validation.

Default: `()`

min `integer` or `none`

If not none, the minimum array length that satisfies the validation. **MUST** be a positive integer. The program is **ILL-FORMED** if `min` is greater than `max`.

Default: `none`

max `integer` or `none`

If not none, the maximum array length that satisfies the validation. **MUST** be a positive integer. The program is **ILL-FORMED** if `max` is less than `min`.

Default: `none`

length `integer` or `auto`

If not auto, the exact array length that satisfies validation. **MUST** be a positiive integer. The program **MAY** be **ILL-FORMED** is concurrently set with either `min` or `max`.

Default: `auto`

custom `function` or `none`

If not none, a function that, if itself returns none, will produce the error set by `custom-error`.

Default: `none`

custom-error `string` or `none`

If set, the error produced upon failure of `custom`.

Default: `auto`

transform `function`

a mapping function called after validation.

Default: `it=>it`

..args `schema` or `none`

Variadic positional arguments of length `0` or `1`. **SHOULD** not contain named arguments. If no arguments are given, schema defaults to array of `any()`

- dictionary()

dictionary

Valkyrie schema generator for dictionary types

Parameters

dictionary(..args: `schema`) -> `schema`

> **..args** `schema`
>
> Variadic named arguments, the values for which are schema types. **MUST** not contain positional arguments.

- either()

either

Valkyrie schema generator for objects that can be any of multiple types.

Parameters

either(..options: `schema`) -> `schema`

..options `schema`

Variadic position arguments for possible types. **MUST** have at least `1` positional argument. Schemas **SHOULD** be given in order of "preference".

- number()
- integer()
- floating-point()
- natural()

number

Valkyrie schema generator for integer- and floating-point numbers

Parameters

```
number(
  name: internal,
  default: integer float none,
  min: integer none,
  max: integer none,
  custom: function none,
  custom-error: string none,
  transform: function,
  types: internal
) -> schema
```

default `integer` or `float` or `none`

Default value to set if none is provided. **MUST** respect all other validation requirements.

Default: `none`

min `integer` or `none`

If not none, the minimum value that satisfies the validation. The program is **ILL-FORMED** if `min` is greater than `max`.

Default: `none`

max `integer` or `none`

If not none, the maximum value that satisfies the validation. The program is **ILL-FORMED** if `max` is less than `min`.

Default: `none`

custom `function` or `none`

If not none, a function that, if itself returns none, will produce the error set by `custom-error`.

Default: `none`

custom-error `string` or `none`

If set, the error produced upon failure of `custom`.

Default: `auto`

transform `function`

a mapping function called after validation.

Default: `it=>it`

integer

Specialization of `number()` that is only satisfied by whole numbers. Parameters of `number()` remain available for further requirments.

Parameters

integer()

floating-point

Specialization of `number()` that is only satisfied by floating point numbers. Parameters of `number()` remain available for further requirments.

Parameters

floating-point()

natural

Specialization of `integer()` that is only satisfied by positive whole numbers. Parameters of `number()` remain available for further requirments.

Parameters

natural()

- string()
- email()
- ip()

string

Valkyrie schema generator for integer- and floating-point numbers

Parameters

```
string(
  name: internal,
  default: string,
  min: integer none,
  max: integer none,
  length: integer auto,
  includes: array string none,
  starts-with: string regex none,
  ends-with: string regex none,
  pattern: string regex none,
  pattern-error: string auto,
  transform: function
) -> schema
```

default `string`

Default value to set if none is provided. **MUST** respect all other validation requirements.

Default: `none`

min `integer` or `none`

If not none, the minimum string length that satisfies the validation. **MUST** be a positive integer. The program is **ILL-FORMED** if `min` is greater than `max`.

Default: `none`

max `integer` or `none`

If not none, the maximum string length that satisfies the validation. **MUST** be a positive integer. The program is **ILL-FORMED** if `max` is less than `min`.

Default: `none`

length `integer` or `auto`

If not auto, the exact string length that satisfies validation. **MUST** be a positiive integer. The program **MAY** be **ILL-FORMED** is concurrently set with either `min` or `max`.

Default: `auto`

includes `array` or `string` or `none`

If set, a coerced array of required strings that are required to pass validation.

Default: `()`

starts-with `string` or `regex` or `none`

If set, a string or regex requirement that the string **SHOULD** start with in order to pass validation.

Default: `none`

ends-with `string` or `regex` or `none`

If set, a string or regex requirement that the string **SHOULD** end with in order to pass validation.

Default: `none`

pattern `string` or `regex` or `none`

If set, a string or regex requirment over the entire string that **SHOULD** be matched in order to pass validation.

Default: `none`

pattern-error `string` or `auto`

If set, the error thrown if `pattern` is not satisfied.

Default: `auto`

transform `function`

A transformation applied after successful validation.

Default: `it=>it`

email

A specialization of string that is satisfied only by email addresses. **Note**: The testing is not rigourous to save on complexity.

Parameters

email()

ip

A specialization of string that is satisfied only by valid IP addresses. **Note**: The testing **IS** strict.

Parameters

ip()

- tuple()

tuple

Valkyrie schema generator for an array type with positional type reqruiements. If all entries have the same type, see `array()`.

Parameters

```
tuple(
  name: internal,
  ..args: schema
) -> schema
```

..args `schema`

Type requirments. Position of argument **MUST** match position of entry in tuple being validated. **SHOULD** not contain named arguments.

III Advanced usage

This section covers topics than the novice-to-intermediate users are unlikely to need to know. If you are looking for information on something you want to achieve, and have not found information somewhere within this guide on how to do so, please submit an issue or pull request on GitHub so that further documentation can be added.

Internal functions

These functions are available under `z.advanced`.

assert-base-type

Asserts the presence of the magic number on the given object.

Parameters

```
assert-base-type(
  arg: any,
  scope: scope
) -> none
```

scope `scope`

Array of strings containing information for error generation.

Default: `("arguments",)`

assert-base-type-array

Asserts the presence of the magic number on an array of object.

Parameters

```
assert-base-type-array(
  arg: any,
  scope: scope
) -> none
```

scope `scope`

Array of strings containing information for error generation.

Default: `("arguments",)`

assert-base-type-dictionary

Asserts the presence of the magic number in a dictionary of object.

Parameters

```
assert-base-type-dictionary(
  arg: any,
  scope: scope
) -> none
```

scope `scope`

Array of strings containing information for error generation.

Default: `("arguments",)`

assert-base-type-arguments

Asserts the presence of the magic number in an argument of object.

Parameters

```
assert-base-type-arguments(
  arg: any,
  scope: scope
) -> none
```

scope `scope`

Array of strings containing information for error generation.

Default: `("arguments",)`

base-type

Schema generator. Provides default values for when defining custom types.

Parameters

```
base-type()
```

Defining a schema generator for a new type

The Typst package ecosystem is large and evergrowing. Eventually, someone, somewhere, will want to validate a type or structure that has never been seen before. If this describes your situation, the following guide may be of use. This section covers different ways complicated types can be defined.

Type specialization - Novice

It may be the case that your type is simply a narrowing of an already-defined type. In such cases, it may be easy to add a validator for your code. For example, to create a validator for numbers between 5 and 10, you could so as as follows:

```
#let specific-number = z.number.with(min: 5, max: 10)
```

Type specialization - Intermediate

If the above method is not sufficient to accurately describe your type, then the custom argument (described above) may be suitable

```
#let specific-number = z.number.with(
    custom: ( it ) => { return (it > 5 and it < 10)},
    custom-error: "Value was incorrect"
)
```

Type specialization - Advanced

If the above doesn't work, but would if you had access to information that would otherwise be hidden inside the schema type-like object, then bootstrapping it may be an avenue to explore

```
#let specific-number(..args) = {
    let internals = z.number(..args)
    return (:..internals,
        // Configure values manually, perhaps override functions.
        // Check source code of schema generator being bootstrapped.
    )
}
```

Type specialization - Wizard

For the most advanced types, creating a schema generator from scratch may be the only way (though this definitely is the last stop, this method should cover all cases). To do so, simple define a function that returns a schema-like dictionary.

```
#let tuple(my-args, ...) = {

  return (:..z.advanced.base-type(), // Shorthand for the definitions shown below. If you do not modify a function, you may aswell omit it and have it set to its default by base-type()
    valkyrie-type: true, // Magic number

   name: "my-type", // Member sometimes used by other classes when they report a failed validation

    // Helper function, generally called by validate()
    assert-type: (self, it, scope:(), ctx: ctx(), types: ()) => {
      if ( type(it) not in types){
        (self.fail-validation)(self, it, scope: scope, ctx: ctx,
```

```
          message: "Expected " + joinWithAnd(types, ", ", " or ") + ". Got " + type(it))
        return false
      }
      return true
    },

   // Do your validation here. Call fail-validation() if validation failed. Generally,
return none also.
    validate: (self, it, scope: (), ctx: (:)) => it,

    // Customize the mode of failure here
    fail-validation: (self, it, scope: (), ctx: (:), message: "") => {
      let display = "Schema validation failed on " + scope.join(".")
      if ( message.len() > 0){ display += ": " + message}
      ctx.outcome = display
      if ( not ctx.soft-error ) {
        assert(false, message: display)
      }
    }
  )
}
```