Differences from spec.alpha

Alex Miller edited this page Dec 17, 2019 · 20 revisions

A summary of differences between spec.alpha and spec 2 (aka alpha.spec). Everything is subject to change as spec 2 is a work in progress.

Symbolic specs

spec 1 defined a language of spec forms (like `s/and`) to define specs. These specs were implemented as macros taking both evaluated objects (symbols or functions) and nested spec forms. The api call `s/form` would produce a symbolic spec from a spec object. One downside of this approach is that the reliance on macros made it challenging to programmatically construct spec forms without using eval or additional macros.

In spec 2, we are more strictly separating the worlds of symbolic specs and spec objects. Symbolic specs consist only of:

- Spec forms (lists/seqs), with spec op in function position, composed of other symbolic specs
- Qualified keywords (names that can be looked up in the registry)
- Qualified symbols (predicate function references)
- Sets of constant values (an enumeration)

The function `s/resolve-spec` takes symbolic specs and returns spec objects (extensions of the Spec protocol) for runtime use. `s/form` is the inverse operation, which takes a spec object and returns a symbolic spec.

The spec forms are themselves macros, which expand to a call to `resolve-spec` on themselves. The only modification made is that spec form macros accept symbols that are not fully-qualified and will qualify ("explicate") them in the namespace context where they are invoked (at compile-time). This is a convenience for people writing spec forms.

The spec registry is a stateful runtime construct that provides a mapping from spec names (either qualified keywords or qualified symbols) to spec objects. A new function `s/register` has been added that adds a mapping from name to spec object. `s/def` is a helpful wrapper macro for `s/register` that understands how to interpret all kinds of symbolic specs (not just spec forms), including symbols and sets, which will be wrapped in the helper `s/spec` spec op.

The spec API functions (`valid?`, `conform`, `explain`, etc) now accept only keywords (spec names), and spec objects. Spec 1 accepted things like function objects (evaluated symbols or anonymous functions) as well - those now must be explicitly wrapped in `s/spec` forms.

Spec additions and changes

Nonflowing s/and- (new)

In spec 1, `s/and` will flow the conformed value through validation/conforming/unform (in reverse), etc. In spec 2, we've added `s/and-` for non-flowing and:

- `s/valid` - validates all preds on original value
- `s/conform` - returns `s/conform` of first pred on value (+ validates other preds)
- `s/unform` - returns `s/unform` of first pred on value
- `s/gen` - gens from first pred, filters only those valid on all subsequent preds (same as `s/and`)

```
(s/def ::x (s/and- (s/cat :i1 int? :i2 int?) #(apply distinct? %)))
(s/valid? ::x [1 2])  ;; true
(s/valid? ::x [1 1])  ;; false

(s/conform ::x [1 2])
;;=> {:i1 1, :i2 2}
(s/explain ::x [1 1])
;; [1 1] - failed: (apply distinct? %) spec: :user/x

(gen/sample (s/gen ::x))
;;=> ((0 2) (-1 0) [0 -1] (-2 -1) (5 -5) (-8 15) (-3 1) [-1 -24] [-26 -1] [-116
-1])

(s/unform ::x {:i1 1, :i2 2})
;;=> (1 2)
```

We may rename these functions in spec 2: `s/and` *will be nonflowing and* `s/and->` *will be flowing.*

s/cat

s/cat continues to accept all sequential collections but will now generate both sequences and vectors. You can combine s/cat with s/and- to narrow this further:

```
(s/def ::vcat (s/and- (s/cat :i int?) vector?))
(s/valid? ::vcat [1]) ;; true
(s/valid? ::vcat '(1)) ;; false

(s/conform ::vcat [1])
;;=> {:i 1}

(s/explain ::vcat '(1))
;; (1) - failed: vector? spec: :user/vcat

(gen/sample (s/gen ::vcat))
;;=> ([-2] [-1] [-1] [0] [-2] [17] [-7] [2] [-7] [-32])
```

Nested regex contexts

Regex specs (`cat`, `alt`, `*`, `+`, `?`, etc) combine to describe a single sequential collection. If the collection contains a nested collection, something needs to be inserted to prevent that combination. In spec 1, this was done with the `s/spec` macro. In spec 2, a new `s/nest` operation serves this purpose and is used only for this.

Creating Specs Programmatically

The new `s/resolve-spec` functional entry point can be used to construct spec objects from spec forms without invoking the spec op macros or using eval:

```
(require '[clojure.alpha.spec :as s])

(defn or-of
  "Make or spec where tags match names"
  [& spec-names]
  (let [tag-names (->> spec-names (map name) (map keyword))]
    (s/resolve-spec (cons `s/or (interleave tag-names spec-names)))))

(s/def ::a int?)
(s/def ::b keyword?)
(s/conform (or-of ::a ::b) 100)
```

```
;; [:a 100]

;; Register using the functional s/register, not s/def
(s/register ::x (or-of ::a ::b))
(s/form ::x)
;; (clojure.alpha.spec/or :a :user/a :b :user/b)
```

Implementing Custom Specs

Simple spec ops

It's common to need a parameterized custom spec op and these can now easily be created with `s/defop`. Ops created with `s/defop` are defined by a parameterized specs, have the form you'd expect, and can optionally provide a custom generator (otherwise will use gen from the spec provided). They conform/unform based on the spec definition.

```
(s/defop bounded-string
  "Specs a string with bounded size (<= min (count string) max)"
  [min max]
  (s/and string? #(<= min (count %) max)))

user=> (s/def ::first-name (bounded-string 1 20))
:user/first-name
user=> (s/form ::first-name)
(user/bounded-string 1 20)
user=> (s/conform ::first-name "Homer")
"Homer"
user=> (s/explain ::first-name "")
"" - failed: (<= 1 (count %) 20) spec: :user/first-name
user=> (gen/sample (s/gen ::first-name))
("q" "q" "0" "O" "vABF" "tk7Dh" "b" "vGn8t7" "Zvaa" "Ycv6M8QBq")
```

Full spec ops

Custom spec ops can be created and installed with a two-step process. First, create a spec op macro that explicates (fully-qualifies) a form and invokes the functional interface:

```
(defmacro my-spec
  [& opts]
  `(s/resolve-spec '~(s/explicate (ns-name *ns*) `(my-spec ~@opts))))
```

Second, calls to `resolve-spec` get routed (via the spec name) to the multimethod `create-spec`

that actually creates the spec by reifying the Spec protocol:

```
(defmethod create-spec 'my-ns/my-spec
  [[_ & opts]]
  (reify Spec
    ...))
```

Because this requires implementing the full Spec protocol, this is a significantly higher effort and something that should only be used for new spec ops that can't easily be created as combinations of the core ops.

Closed spec checking

Spec is primarily concerned with an "open" approach to map validation. s/keys (and now s/schema) defined a set of attributes that can co-occur. s/keys (and now s/select) can be used to specify requirements of maps, but not negative constraints (maps can *only* contain these keys, and no others). Allowing for open maps allows specs to evolve over time. There are more changes coming to better talk about the requires/provide split at the function level.

However, there are a variety of situations where it is useful to do closed spec checking - calling an API known to be fragile in the face of unexpected attributes, verifying user inputs according to a specification, or even just checking for attribute typos. In these cases, you can now enable "closed spec" checking where unspecified keys fail validation. All of the conforming api calls (`s/valid?`, `s/conform`, `s/explain` and similar) now optionally take an additional settings argument. The settings are a map of setting to configuration for that setting.

In the case of closed specs, use `:settings` and a set of schema specs to close:

```
(s/def ::f string?)
(s/def ::l string?)
(s/def ::s (s/schema [::f ::l]))

;; "extra" keys are ok normally - open maps are the default
(s/valid? ::s {::f "Bugs" ::l "Bunny" ::x 10})
;;=> true

;; but closed spec checking can be more restrictive
(s/valid? ::s {::f "Bugs" ::l "Bunny" ::x 10} {:closed #{::s}})
;;=> false

(s/explain ::s {::f "Bugs" ::l "Bunny" ::x 10} {:closed #{::s}})
#:user{:f "Bugs", :l "Bunny", :x 10} - failed:
  (subset? (set (keys %)) #{:user/f :user/l})
```

Clojure integration

Because Clojure itself does not know about spec 2, certain integration features will not work as expected (`doc` won't see the registry, error stack reporting may not print the correct failure location during instrumentation, and macros will not be automatically checked).

Pages 3

Find a Page…

Home

Differences from spec.alpha

Schema and select