clojure.spec: a lisp-flavoured type system

@sbelak
simon.belak@gmail.com

Motivation

- Communication (docs are not enough)

Motivation

- Communication (docs are not enough)
- Error handling & reporting leaves a lot to be desired

Motivation

- Communication (docs are not enough)
- Error handling & reporting leaves a lot to be desired
- A lot of computation is encoded with (naked) data (how can we best manipulate and validate)

Motivation

- Communication (docs are not enough)
- Error handling & reporting leaves a lot to be desired
- A lot of computation is encoded with (naked) data (how can we best manipulate and validate)
- Generative (property based) testing and test data generation

Motivation

- Communication (docs are not enough)
- Error handling & reporting leaves a lot to be desired
- A lot of computation is encoded with (naked) data (how can we best manipulate and validate)
- Generative (property based) testing and test data generation
- Manual parsing is common, tedious, and error prone

Motivation

- Communication (docs are not enough)
- Error handling & reporting leaves a lot to be desired
- A lot of computation is encoded with (naked) data (how can we best manipulate and validate)
- Generative (property based) testing and test data generation
- Manual parsing is common, tedious, and error prone

Enter clojure.spec

Parsing with Derivatives

A Functional Pearl

Matthew Might David Darais
University of Utah
might@cs.utah.edu, david.darais@gmail.com

Daniel Spiewak
University of Wisconsin, Milwaukee
dspiewak@uwm.edu

Abstract

We present a functional approach to parsing unrestricted context-free grammars based on Brzozowski's derivative of regular expressions. If we consider context-free grammars as recursive regular expressions, Brzozowski's equational theory extends without modification to context-free grammars (and it generalizes to parser combinators). The supporting actors in this story are three concepts familiar to functional programmers—laziness, memoization and fixed points; these allow Brzozowski's original equations to be transliterated into purely functional code in about 30 lines spread over three functions.

Yet, this almost impossibly brief implementation has a drawback: its performance is sour—in both theory *and* practice. The culprit? Each derivative can *double* the size of a grammar, and with it, the cost of the next derivative.

Fortunately, much of the new structure inflicted by the derivative is either dead on arrival, or it dies after the very next derivative. To eliminate it, we once again exploit laziness and memoization

The derivative of regular expressions [1], if gently tempered with laziness, memoization and fixed points, acts immediately as a pure, functional technique for generating parse forests from arbitrary context-free grammars. Despite—even because of—its simplicity, the derivative transparently handles ambiguity, left-recursion, right-recursion, ill-founded recursion or any combination thereof.

1.1 Outline

- After a review of formal languages, we introduce Brzozowski's derivative for regular languages. A brief implementation highlights its rugged elegance.
- As our implementation of the derivative engages context-free languages, non-termination emerges as a problem.
- Three small, surgical modifications to the implementation (but not the theory)—laziness, memoization and fixed points—guarantee termination. Termination means the derivative can

clojure.spec

Regular expressions (for data)

+

arbitrary predicates

+

data transformations (conformers)

```clojure
(s/def ::quantity pos-int?)
(s/def ::product (partial contains? products))
(s/def ::timestamp (s/and string?
                          (partial re-matches #"\d{4}-\d{2}-\d{2}")
                          (s/conformer time/parse)))

(s/def ::order (s/keys :req [::timestamp ::quantity ::product]))
(s/def ::orders (s/+ ::order))

[...]

(s/def ::user (s/keys :req [::user-id ::orders ::account-age]))
```

“Good design is not about making grand plans, but about taking things apart.”

— R. Hickey

Live programming

lisp runs in the same context it’s written in

spec: a fully interactive à la carte type system?

Destructuring

- Pull apart and name
- Factor out data shape
- Transform

```
(s/def ::timestamp (s/and string?
                          (partial re-matches #"\d{4}-\d{2}-\d{2}")
                          (s/conformer time.format/parse)))

(s/def ::order (s/cat :timestamp ::timestamp
                      :quantity pos-int?
                      :product (partial contains? products)))

(s/def ::visit (s/cat :timestamp ::timestamp
                      :page string?))

(s/def ::event (s/alt :order ::order
                      :visit ::visit))
```

```
spec.demo> (s/conform ::event ["2017-01-17" 2 4])
[:order {:timestamp #object[org.joda.time.DateTime 0x3dabbdee "2017-01-17T00:00:00.000Z"], :quantity 2, :product 4}]
spec.demo> (s/conform ::event ["2017-01-17" "index.html"])
[:visit {:timestamp #object[org.joda.time.DateTime 0x5e51a0f3 "2017-01-17T00:00:00.000Z"], :page "index.html"}]
```

Data macros

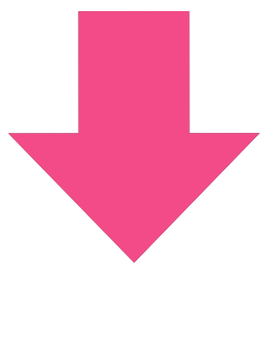
- Recursive transformations into canonical form

- Do more without code macros

* juxt.pro/blog/posts/data-macros.html

```clojure
(where {(any-of :foo :bar) pos?
        :answer 42
        :timetamp [after? 2016]}
       coll)
```

```
(where {(any-of :foo :bar) pos?
        :answer 42
        :timetamp [after? 2016]}
       col)
```



```
(where {[::combinator some-fn
         ::keyfns [#(safe-get % :foo) #(safe-get % :foo)]} pos?
        {::combinator identity
         ::keyfns [#(safe-get % :answer)]} (partial = 42)
        {::combinator identity
         ::keyfns [#(safe-get % :timestamp)]} #(after? % 2016)}
       col)
```

Decomplect validation and encoding

Generative testing

- Can uncover numerical instabilities (can your type system do this?)

- Better mocking in the REPL

Everything is data

(aka. the lisp way)

- generation
- introspection
- parsable errors

Building on top of spec

github.com/arohner/spectrum — static analysis of (spec annotated) Clojure code

github.com/stathissideris/spec-provider — infer spec from examples

github.com/typedclojure/auto-annotation — infer spec from tests

Takeouts

Decomplect everything

Everything should be live and interactive

Blurring the line between environment and work is a powerful idea

Queryable data descriptions supercharge interactive development and are a great building block for automation

Questions

@sbelak
simon@goopti.com

clojure.org/about/spec

matt.might.net/papers/might2011derivatives.pdf

infoq.com/presentations/Mixin-based-Inheritance

realworldclojure.com/the-system-paradigm

juxt.pro/blog/posts/data-macros.html

blog.klipse.tech/clojure/2016/10/02/parsing-with-derivatives-regular.html

indiegogo.com/projects/typed-clojure-clojure-spec-auto-annotations#/

github.com/arohner/spectrum

github.com/stathissideris/spec-provider

purelyfunctional.tv/issues/clojure-gazette-192-composition-is-about-decomposing