Typed Clojure

Tool-assisted spec development

Ambrose Bonnaire-Sergeant
Sam Tobin-Hochstadt
Spec is awesome

- Expressive specification language
- Runtime instrumentation
- Generative testing
- Documentation
- Parsing
Show of hands:
Who’s avoided writing a spec because...?

- Too inexperienced with spec
- Reverse engineering too costly to justify
- Partially wrote spec, but will “fill it in later”
Show of hands:
Who’s updated a spec because...?

- Typo in the spec
- Function/data out of date with spec
We need help writing our specs
Idea

- Spec models how our program currently operates
  - So... **run the program**, and **observe**!
- Use observations to generate specs
Analogy:
Megaphone as a function
Goal:
What is the “spec” for a megaphone?
Step 1: Instrumentation function
Step 1: Instrumentation function
Step 2:
Observe running program

function
Step 2: Observe running program
Step 3:  
Summarize execution
Step 3: Summarize execution
The challenge:
Deriving “good” specs from program execution
Values

:foo

Specs

1 any?  2 keyword?  3 #{:foo}

The observed value

Our choices to spec value
Values

:foo

Specs

1 any?
2 keyword?
3 #{:foo}

Don’t care

Homogeneous

Enumeration
Values

\{1 \ 2, \ 3 \ 4\}

Specs

1 map?

2 (map-of int? int?)

3 (map-of #{1 3} #{2 4})

4 (map-of pos? pos?)

- Instance check
- Homogeneous
- Enumerate
- Constrain
Values

\{::x\ 2, \::y\ 3\}

Specs

1 (map-of keyword? int?)

2 (def ::x int?)
   (def ::y int?)
   (s/keys :req-un [::x ::y])
{::shape :square
  :width 3}

1 (map-of keyword? (or int? keyword?))
2 (def ::shape keyword?)
   (def ::width int?)
   (def ::height int?)
   (s/keys :req-un [:::shape ::width])
3 (defmulti shape :shape)
   (defmethod shape [__] :square
    (s/keys :req-un [:::shape ::width]))
   (multi-spec spec shape :shape)
{:shape :square
 :width 3}
{:shape :rect
 :width 3
 :height 4}

1 (map-of keyword? (or int? keyword?))
2 (def ::shape keyword?)
   (def ::width int?)
   (def ::height int?)
   (s/keys :req-un [:shape ::width]
             :opt-un [:height])
3 (defmulti shape :shape)
   (defmethod shape [__] :square
                (s/keys :req-un [:shape ::width]))
   (defmethod shape [__] :rect
                (s/keys :req-un [:shape ::width
                                ::height])))

(multi-spec shape :shape)

Values

Optional entries

Second tag

Specs
Values

{(defmulti shape :shape)
  (defmethod shape [_] :square
    (s/keys :req-un [:shape :width]
      :opt-un [:inner]))
  (defmethod shape [_] :rect
    (s/keys :req-un [:shape :width :height]
      :opt-un [:inner]))
(s/def ::inner (multi-spec shape :shape))
(multi-spec shape :shape)}
Phew!
Many implicit decisions
How do we make sense of all this?

1. View possible specs as a lattice

2. Observing different values moves along lattice
How do we make sense of all this?

1. View possible specs as a **lattice**

2. Observing different values **moves** along lattice
How do we make sense of all this?

1. View possible specs as a **lattice**

2. Observing different values **moves** along lattice
How do we make sense of all this?

1. View possible specs as a lattice
2. Observing different values moves along lattice
How do we make sense of all this?

1. View possible specs as a **lattice**

2. Observing different values **moves** along lattice
Build lattice using heuristics about Clojure idioms
Observed Values

1 2 3

Chosen Spec

(number? (or double? integer?) double? integer?)
Observed Values

1 2 3
4.5
6.2

Chosen Spec

number?
(or double? integer?)
double?
integer?

Heuristic

Observed doubles
### Observed Values

1 2 3
4.5
6.2
532.23M

### Chosen Spec

- number?
- (or double? integer?)
- double?
- integer?
Observed Values

{:shape :rect ...}

Chosen Spec

any?

(map-of keyword? any?)

(map-of keyword? (or ...))

(multi-spec shape :shape)

(s/keys :req-un [:shape])

Entity map
Observed Values

{:shape :rect ...}
{:shape :circle ...}

Chosen Spec

any?

(map-of keyword? any?)

(map-of keyword? (or ...))

(multi-spec shape :shape)

(s/keys :req-un [:shape])

Tagged Entity map
Observed Values

{::{shape :rect} ...
{::{shape :circle} ...
{::{x 1 :y 2}

Chosen Spec

(map-of keyword? any?)
(map-of keyword? (or ...))
(multi-spec shape :shape)
(s/keys :req-un [:::shape])

Homogeneous map

any?
Observed Values

{:shape :rect ...}
{:shape :circle ...}
{:x 1 :y ...
42 'sym :kw

Chosen Spec

any?

(map-of keyword? any?)

(map-of keyword? (or ...))

(s/keys :req-un [:::shape])

({:shape :circle ...})

Observed non-map data
Demo
Real-world Examples
The good...
Names

```lisp
+(s/fdef expt-int :args (s/cat :base int? :pow int?) :ret int?)
+(s/fdef
  + init
  + :args
  + (s/cat :n int? :s (s/or :nil? nil? :int? int?))
  + :ret
  + (s/coll-of int?))
+(s/fdef
  + count-permutations-from-frequencies
  + :args
  + (s/cat :freqs (s/map-of (s/or :char? char? :int? int?) int?))
  + :ret
  + int?)
```
Simple collection manipulation

(s/fdef
  + lex-permutations
  + :args
  + (s/cat :c (s/coll-of int?))
  + :ret
  + (s/coll-of (s/coll-of int? :into vector??))
Multiple arities

+ (s/fdef
+  + date-time
+  + :args
+  + (s/alt
+  + :1-arg
+  + (s/cat :year int?)
+  + :2-args
+  + (s/cat :year int? :month int?)
+  + :3-args
+  + (s/cat :year int? :month int? :day int?)
+  + :4-args
+  + (s/cat :year int? :month int? :day int? :hour int?)
+  + :5-args
+  + :6-args
Multi-specs

(defmulti op-multi-spec4772367 :op)
(defmethod
  op-multi-spec4772367
  :map
  []
  (s/keys
   :req-un
   [:clojure.core.Typed.unqualified-keys/children
    :clojure.core.Typed.unqualified-keys/env
    :clojure.core.Typed.unqualified-keys/form
    :clojure.core.Typed.unqualified-keys/keys
    :clojure.core.Typed.unqualified-keys/op
    :clojure.core.Typed.unqualified-keys/tag
    :clojure.core.Typed.unqualified-keys/vals]))

(defmethod
  op-multi-spec4772367
  :vector
  []
  (s/keys
   :req-un
   [:clojure.core.Typed.unqualified-keys/children
    :clojure.core.Typed.unqualified-keys/env
    :clojure.core.Typed.unqualified-keys/form
    :clojure.core.Typed.unqualified-keys/keys
    :clojure.core.Typed.unqualified-keys/op
    :clojure.core.Typed.unqualified-keys/tag
    :clojure.core.Typed.unqualified-keys/vals]))

(defmethod
  op-multi-spec4772367
  :var
  []
  (s/keys
   :req-un
   [:clojure.core.Typed.unqualified-keys/children
    :clojure.core.Typed.unqualified-keys/env
    :clojure.core.Typed.unqualified-keys/form
    :clojure.core.Typed.unqualified-keys/keys
    :clojure.core.Typed.unqualified-keys/op
    :clojure.core.Typed.unqualified-keys/tag
    :clojure.core.Typed.unqualified-keys/vals]))
Useful aliases

Extract common map into alias

Usage

Usage
Mutually recursive maps

```
+(defalias
  + T
  + (U
    + '{:T ':false}
    + '{:T ':fun, :params [NameTypeMap], :return T}
    + '{:T ':intersection, :types (Set T))
    + '{:T ':num}))
```

```
+(defalias
  + P
  + (U
    + '{:P ':=, :exp (Set E)}
    + '{:P ':and, :ps (Set P)}
    + '{:P ':is, :exp E, :type T}
    + '{:P ':not, :p P}
    + '{:P ':or, :ps (Set P))}
```

```
+(defalias
  + E
  + (U
    + '{:E ':app, :args (Vec E), :fun E}
    + '{:E ':false}
    + '{:E ':if, :else E, :test E, :then E}
    + '{:E ':lambda, :arg Sym, :arg-type T, :body E}
    + '{:E ':var, :name Sym}))
```
Mutually recursive maps
The bad...
Lost kw arguments

```clojure
+ (s/def  
  + lex-partitions-H  
  + :args  
  + (s/alt  
  + :1-arg  
  + (s/cat :N int?)  
  + :5-args  
  + (s/cat  
  + :N  
  + int?  
  + :rest-arg-0  
  + #{:min}  
  + :rest-arg-1  
  + int?  
  + :rest-arg-2  
  + #{:max}  
  + :rest-arg-3  
  + int?)

: min kw arg

: max kw arg
```
Over-specificity
The ugly...
No rest
arg inference

13 args

6 args

13 args
The future
Polymorphic types

19 (t/ann identity [t/Int :-> t/Int])
20 (comment (t/ann identity (t/All [x] [x :-> x])))
21 (comment (t/ann identity (t/All [x] [x :-> x])))
Polymorphic types

Fragments of polymorphic types

23 (t/ann mymap
24
25 [[(t/U t/Int t/Sym) :-> (t/U t/Str t/Int)]
26 (t/Vec (t/U t/Int t/Sym))
27 :->
28 (t/Coll (t/U t/Str t/Int)))]]
29 (comment (t/ann mymap (t/All [x] [[x :-> ?] (t/Vec x) :-> ?])))
30 (comment (t/ann mymap (t/All [x] [[x :-> ?] '[[? ? x] :-> ?]]))
31 (comment (t/ann mymap (t/All [x] [[? :-> x] ? :-> (t/Coll x)]))
32 (comment (t/ann mymap (t/All [x] [[x :-> ?] '[[x ? ?] :-> ?]]))
33 (comment (t/ann mymap (t/All [x] [[x :-> ?] '[[? x ?] :-> ?]]))
34 (comment (t/ann mymap (t/All [x] [[? :-> x] ? :-> (t/Coll x)]))
35 (comment (t/ann mymap (t/All [x] [[x :-> ?] '[[x] :-> ?]]))
Questions, Demos, and magic tricks?
Thanks
Automatic Annotations for Typed Clojure + clojure.spec

This page summarises my work on automatic annotation generation.

Library annotations

Here I will list a bunch of libraries we have generated annotations for. They don't type check, but the idea is they're very close--- and with good alias names! Last updated: 3rd April 2017

- startrek-clojure  Generated core.typed Manually type checked diff clojure.spec
- math.combinatorics  Generated core.typed Manually type checked diff clojure.spec
Inference results via side effects

(point 1 2)
; ['point {:dom 0}] : Long
; ['point {:dom 1}] : Long
; ['point :rng (key :x)] : Long
; ['point :rng (key :y)] : Long
{:x 1
 :y 2}
Step 1: \( \text{gen} \Gamma : r \rightarrow \Gamma \)

1) Generate naive type environment from dynamic inference results

```plaintext
; ['point{:dom 0}] : Long
; ['point{:dom 1}] : Long
; ['point:rng (key :x)] : Long
; ['point:rng (key :y)] : Long
```

point : [Long Long -> '{:x Long :y Long}']
Step 2: squashLocal : $\Gamma \rightarrow \Delta$

2) Create local recursive types ("vertically")

\[
\begin{align*}
f : & \\
g : & \\
h : & \\
b : & 
\end{align*}
\]
Step 2: squashLocal : \( \Gamma \rightarrow \Delta \)

2) Create local recursive types ("vertically")
Step 3: $\text{squashGlobal} : \Delta \to \Delta$

3) Merge possibly-recursive types globally ("horizontally")
Step 3: squashGlobal : Δ → Δ

3) Merge possibly-recursive types globally ("horizontally")
Instrumentation

(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :val (:op expr)) (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...)))
Instrumentation

(ann emit [Expr -> String])

(defn emit [expr]
  (cond
    (= :val (:op expr)) (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...))

(ann emit [Expr -> String])
(defn emit [expr]
  (cond
   (= :val (:op expr)) (emit-val expr)
   (= :list (:op expr)) (emit-list expr)
   ...)))