Typed Clojure in Theory and Practice

Ambrose Bonnaire-Sergeant

Clojure

- Dynamic typing ¯_(ツ)_/¯
- Functional style *(map f (filter g …))*
- Lisp-style Macros *(for […])*
- Immutable data structures 
- Hosted on JVM **Java**

Immutable maps

Global function

```
(defn point [x y] {:x x, :y y})
```

Global definition

```
(def p (point 1 2))
;=> {:x 1 :y 2}
```

Assoc-iate entry

```
(assoc p :x 3)
;=> {:x 3 :y 2}
```

Dissoc-iate entry

```
(dissoc p :y)
;=> {:x 1}
```

Lookup entry

```
(get p :x)
;=> 1
```

Java interop

Java

```
(defn upper-case [s]
  (when s
    (.toUpperCase s)))

(upper-case nil)
;=> nil

(upper-case “abc”)
;=> “ABC”
```

Null test → s

Method call → (.toUpperCase s)

Macros

Macro definition

```
(defmacro when [t body]
  `(if ~t ~body nil))
```

"Thread first" macro

```
(-> {}                ; {}
    (assoc :x 3)      ; {:x 3}
    (assoc :y 4))     ; {:x 3 :y 4}
;=> {:x 3 :y 4}
```

"Thread last" macro

```
(->> [1 2 3 4]           ; [1 2 3 4]
     (map inc)           ; (2 3 4 5)
     (filter even?))     ; (2 4)
;=> (2 4)
```

Higher-order functions

Update map entry

```
(-> {:ms 0}
    (update :ms inc))
;=> {:msg 1}
```

Create Mutable atom

```
(def tick (atom {:ms 0}))
```

Atomic swap

```
(swap! tick update :ms inc)
; {:ms 1}
```

Multimethods

Define multimethod

Dispatch on :op entry

```
(defmulti subst
  ‘‘Apply substitution s on expression m.’’
  (fn [m s] (:op m)))
```

“if” case

```
(defmethod subst :if [m s]
  (-> m
      (update :test subst s)
      (update :then subst s)
      (update :else subst s)))
```

“var” case

```
(defmethod subst :var [m s]
  (-> m
      (update :name #(or (get s %) %))))
```

Transducers

Transducers are composable, algorithmic transformations

Transducer definition

```
(def add-then-filter
  (comp (map inc)
        (filter even?)))
```

Transducer usage

```
(sequence add-then-filter [1 2 3 4])
;=> (2 4)
```

Clojure's Runtime verification

	Clojure.spec	
Heterogeneous maps	✓	
Top-level Functions	✓	
Polymorphic functions	✗	**_Better suited for static analysis_**
Transducers	✗	
Asynchronous Channels	✗	
Multimethods	✗	

Clojure

+

Optional Type system

=

Typed Clojure

Typed Clojure

- Bidirectional Type Checking
- Check idiomatic Clojure code
- Heterogeneous Maps
- Occurrence typing (flow sensitive)
- Prevents Null-pointer exceptions

Thesis Statement

Typed Clojure is a sound and practical optional type system for Clojure.

- Typed Clojure is an **optional type system for Clojure**.
- Typed Clojure is **sound**.
- Typed Clojure is **practical**.

Typed Clojure is a sound and practical optional type system for Clojure.

- Typed Clojure is an **optional type system for Clojure**.
  - Target idiomatic Clojure code
  - Type checking is opt-in
- Typed Clojure is **sound**.
  - Formal model of core type system features
  - Prove type soundness for model
- Typed Clojure is **practical**.
  - Type system supports actual Clojure usage patterns.
  - Address user feedback.

Thesis Statement

Typed Clojure is a sound and practical optional type system for Clojure.

Part 1: Initial design & Evaluation

Bidirectional Type Checking

```
(ann upper-case [(U Str nil) -> (U Str nil)])
(defn upper-case [s]
  (when s
    (.toUpperCase s)))
```

Top-level annotations

Type-based control flow

Explicit null type

```
(ann upper-case [(U Str nil) -> (U Str nil)])
(defn upper-case [s]
  (when s                Str
    (.toUpperCase s)))
```

Refined type via occurrence typing

Avoiding null-pointer exceptions

```
(ann upper-case [(U nil Str) -> (U nil Str)])
(defn upper-case [s]
  (when s
    (.toUpperCase s)))
```

(U nil Str)

Str

Evaluation
62/62 methods avoid null-pointer exceptions

Part 1: Initial design & Evaluation (completed)

- **Theory**: We formalize Typed Clojure, including its characteristic features like hash-maps, multimethods, and Java interoperability, and prove the model type sound. ✓
- **Practice**: We present an empirical study of real-world Typed Clojure usage in over 19,000 lines of code, showing its features correspond to actual usage patterns. ✓
- Published: “Practical Optional Types for Clojure”, Ambrose Bonnaire-Sergeant, Rowan Davies, Sam Tobin-Hochstadt; ESOP 2016 ✓

Thesis Statement

Typed Clojure is a sound and practical optional type system for Clojure.

Part 1: Initial design & Evaluation

Part 2: Automatic Annotations

Annotations needed

```clojure
(ann point [Long Long -> Point])
(defn point [x y]
  {:x x
   :y y})
```

Top-level typed bindings

Untyped libraries

```clojure
(ann clojure.string/upper-case [Str -> Str])
```

Runtime Inference

```clojure
(def forty-two 42)
```

$$\Gamma = \{\texttt{forty-two} : \texttt{Long}\}$$

```clojure
(def forty-two
  (track 42 ['forty-two]))
```

```clojure
; Inference result:
; ['forty-two] : Long
(def forty-two 42)
```

Part 2: Automatic Annotations (in progress)

- **Theory**: We design and formalize an approach to automatically generating top-level type annotations based on example executions.
- **Practice**: We implement and evaluate our algorithm on real Clojure programs. We measure the reduction in the human annotation burden with an empirical study on the number of manual changes needed to type check a program.
- To be submitted: PLDI 2019 (Fall 2018)

Thesis Statement

Typed Clojure is a sound and practical optional type system for Clojure.

Part 1: Initial design & Evaluation
Part 2: Automatic Annotations
Part 3: Support checking more programs

Anonymous functions

Need annotation!

Hard to check

```
(let [f (fn [a] (inc a))]
  (f 1))
```

Anonymous functions

Need annotation!

Hard to check

```
(let [f (fn [a] (inc a))]
  (f 1))
```

Delay check to occurrences

Easier to check

```
(let [f …]
  ((fn [a] (inc a)) 1))
```

Int

Polymorphic Higher-order functions

Hard to check

```
(ann inc-val ['{:val Int} -> '{:val Int}])
(defn inc-val [m]
  (update m :val (fn [v] (inc v))))
```

Polymorphic function cannot propagate information to function arguments (must check arguments ***before*** *solving polymorphic variables)*

Need type!

Polymorphic Higher-order functions

Hard to check

```
(ann inc-val ['{:val Int} -> '{:val Int}])
(defn inc-val [m]
  (update m :val (fn [v] (inc v))))
```

```
(deftyperule update [m k f]
  `(assoc ~m ~k (~f (get ~m ~k))))
```

Polymorphic Higher-order functions

Hard to check

```
(ann inc-val [‘{:val Int} -> ‘{:val Int}])
(defn inc-val [m]
  (update m :val (fn [v] (inc v))))
```

Easier to check

```
(deftyperule update [m k f]
  `(assoc ~m ~k (~f (get ~m ~k))))
```

```
(ann inc-val [‘{:val Int} -> ‘{:val Int}])
(defn inc-val [m]
  (assoc m :val ((fn [v] (inc v)) (get m :val)))
```

Apply type rule

Int ✓

Part 3: Support checking more programs (in progress)

- **Type checking interleaved with expansion:** We motivate and describe how to convert Typed Clojure from a type system that only checks fully expanded programs to one that incrementally checks partially expanded programs, and present an implementation.
- **Extensible type rules**: We describe and implement an extensible system to define custom type rules for usages of top-level functions and macros and study how they improve the inference of core Clojure idioms.
- **Symbolic analysis:** We describe and implement symbolic evaluation strategies for Clojure programs and study how many more programs can be checked.

Thesis Statement

Typed Clojure is a sound and practical optional type system for Clojure.

Part 1: Initial design & Evaluation
Part 2: Automatic Annotations
Part 3: Support checking more programs
(Backup Part 3: Automatic Annotations for clojure.spec)

Backup plan: Automatic Annotations for clojure.spec

Repurpose automation technology: We describe how to automatically generate clojure.spec annotations ("specs") for existing programs by reusing most of the the infrastructure for automatic Typed Clojure annotations. We present a formal model of clojure.spec (an existing and popular runtime verification tool for Clojure) and implement the model in Redex. ✓

Test effectiveness of Annotation tool: Ensure high quality specs are generated, and automatically test over hundreds of projects.

Study how Clojure is used in real projects: We conduct a study of general Clojure idioms and practices by generating, enforcing, and exercising specs across hundreds of projects, as well as analyzing design choices in Typed Clojure's type system, clojure.spec's features, and our automatic annotation tool.

Timeline

August 2018	Finish formal model of Annotation Tool
Sept-Oct 2018	Carry out Auto Annotation experiments
Nov 2018	Submit PLDI paper for Auto Annotations
Dec 2018	Improve & evaluation Extensible typing rules
Jan-May 2019	Write dissertation
June 2019	Defend

Thanks