What if Type Systems were more like Linters?

Ambrose Bonnaire-Sergeant
Me

• A Practical Optional Type System for Clojure (2012)
• Typed Clojure Indiegogo Campaign
• @ambrosebs
Pluggable, Optional Type Systems

- Do not affect program semantics
- Same tooling
- Opt-in
- Combinations
Clojure

- Lisp dialect
- Dynamically typed
- Hosted (JVM, JavaScript, CLR)
- Immutability
Clojure Syntax

(f a1 a2 a3)

Operator

Arguments
Typed Clojure
An optional type system for Clojure
Goals

• Type checker as a Library
• Understand common Clojure style
• Sound type checking
Type System

- Does not affect runtime semantics
- Statically sound
- Explicitly typed with local inference
Traditional Type Systems

Hack → Type Check → Run

Tuesday, 22 April 14
Pluggable, Optional Type Systems

Hack → Type Check → Run
Simple Types

'a

; clojure.lang.Symbol

1

; java.lang.Long

<some-expression>

; Any

(throw e)

; Nothing
Simple Types

(fn> [a :- Number] (inc a))
; [Number -> Number]
Immutable Collections

\[
[1 \ 'a : b]
; \ (Vec \ (U \ Number \ Keyword \ Symbol))
\]

\{
1 2, 3 4
; \ (Map \ Number \ Number)
\}
Type Aliases

(def-alias MyName
   "Optional docstring"
   Symbol)
Unions

• Ad-hoc/untagged unions
• Model data flow common in dynamic languages
Unions

(if c?
  'sym
  1.2)
; (U Number Symbol)
Control Flow

- Need to eliminate union members
- *Occurrence typing* understands common control flow
Eliminating Unions

(let [a (if c? 'a 1)]
  (if (symbol? a)
      (name a)
      (inc a)))
Inline Assertions

(let [a (if c? 'a 1)]
  (assert (number? a))
  (inc a))
Data

- Clojure emphasises *data*
- Maps, vectors, lists
- Often implied structure
  - Typed Clojure understands common patterns
Maps

- Heterogeneous maps
- Optional keys
- Partial maps
- Common operations
  - add/remove keys, merge
Creating Maps

{:a 1, :b 1.2, :c 'a}
; (HMap :mandatory {:a Int,
; :b Number,
; :c Symbol}
; :complete? true)
Modifying Maps

(assoc {:a 1} :b 1.2)
; (HMap :mandatory {:a (Value 1),
;                   :b (Value 1.2)}
;       :complete? true})

(dissoc {:a 1} :a)
; (HMap :complete? true)
Lookup Maps

(:a {:a 1})
; Number

(:a (if c?
    {:a 1}
    {:b 2}))
; (U nil Number)
Merging Maps

(merge (if c?
    {:a 1, :c 'a}
    {:b 2})
{:c 3})

; (HMap :mandatory {:c Number}
;         :optional {:a Number
;                     :b Number}
;         :complete? true)
Occurrence Typing + Maps

(let [a {:a
  {:b (if c? 1 'a)}}]
  (if (number? (-> a :a :b))
    (inc (-> a :a :b))
    (name (-> a :a :b))))
(def.alias Expr
(Rec [Expr]
  (U '{:op (Value :if)
      :test Expr, :then Expr
      :else Expr}
    '{:op (Value :do)
      :exprs (Coll Expr)}
    ...)))

(defmulti parse :op)
(defmethod parse :if
  [{:keys [test then else]}] ...)
(defmethod parse :do
  [{:keys [exprs]}] ...)

Maps + Unions
Java Interop

- Pessimistic by default, overridable by programmer
- Understands Java invariants
Functions

- Ordered intersections (like TypeScript)
- An ordered list of arities that represents a function
Functions

(ann foo [Number -> Number])
(defn foo [n] (inc n))
(def-alias NumSym (U Number Symbol))
(def-alias NumStr (U Number String))

(ann foo2 (Fn [Number -> Number]
               [Symbol -> String]
               [NumSym -> NumStr]))

(defn foo2 [n]
  (if (number? n)
      (inc n)
      (name n))
)
Keyword Arguments

(ann kwarg [& :optional {::kw Number} -> Any])
(defn kwarg [& {:keys [::kw]}]
    ; ::kw :- (U nil Number)
    ...
)

(foo2 ::kw 1)
Polymorphism

- Polymorphic Functions
- Bounded polymorphism
Simple Polymorphism

(ann id (All [x] [x -> x])
(defn id [a] a)
Bounded Polymorphism

(ann add-a (All [[x :< (Map Any Any)]
                    [x -> (Assoc x `:a Number)]])
(defn add-a [m]
             (assoc m :a 1))

(inc (:a (add-a {:b 1})))
(inc (:b (add-a {:b 1})))
Dotted Polymorphism

; Infinite number of arities.
; Dotted polymorphism defines a template that covers all valid usages.
(map + [1 2 3])
(map + [1 2 3] [4 5 6])
(map + [1 2 3] [4 5 6] [7 8 9])
(map + [1 2 3] [4 5 6] [7 8 9] [10 11 12])
Macros

• macros + types = yum!
• Roll your own syntax
Pretty def

(defmacro def [nme _ t init]
  `(do (ann ~nme ~t)
       (core/def ~nme ~init))

(def v :- Symbol 'a)
; Same as:
; (do (ann v Symbol)
;     (def v 'a))

Tuesday, 22 April 14
Conclusion

• Pluggable type systems don’t change semantics

• They provide layers of verification on top of a language

• Your favourite language?