Typed Clojure

Practical Optional Types for Clojure

Ambrose Bonnaire-Sergeant*
Rowan Davies**
Sam Tobin-Hochstadt*

*

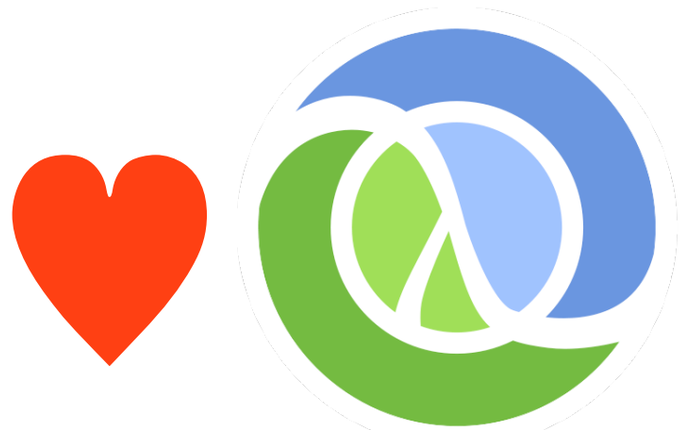
Omnia Team, Commonwealth Bank of Australia**

Confession...

I 

Clojure

Lot of reasons...

I 

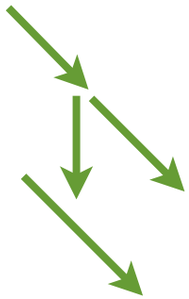
Clojure

Why?

Why I love Clojure!

Immutable data structures

Local reasoning



Hosted on JVM

Java

Lisp-style Macros

‘()

Dynamic typing

¯_(ツ)_/¯

I think …

Clojure encourages disciplined programming

Question

How can we evaluate this claim?

But first…

What does

Clojure

look like?

Immutable maps

Global function

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

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

Immutable maps

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

Immutable map literal

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

Immutable maps

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

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

Prefix notation

Immutable maps

```
(defn point [1 2]
  {:x 1
   :y 2})
```

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

Prefix notation

Immutable maps

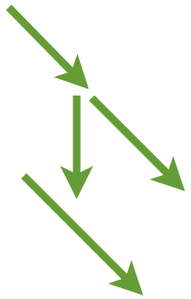
```
(defn point [x y]
  {:x x
   :y y})
```

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

```
(inc (:x (point 1 2)))
;=> 2
```

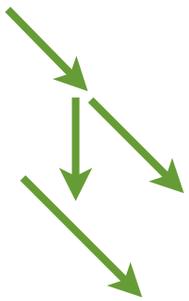
Keyword lookup

Local Reasoning



```
(defn flip [n]
  (if (string? n)
    (- (Long/parseLong n))
    (- n)))

(flip 42)
;=> -42

(flip "42")
;=> -42
```

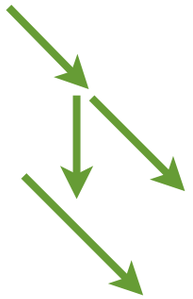
Local Reasoning



```
(defn flip [n]
  (if (string? n)
    (- (Long/parseLong n))
    (- n)))

(flip 42)
;=> -42

(flip "42")
;=> -42
```

Informs branches
the type of n

Local Reasoning



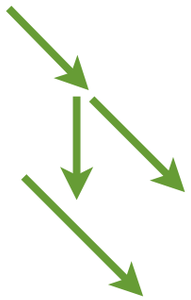
```
(defn flip [42]
  (if (string? 42)
    (- (Long/parseLong n))
    (- 42)))
```

```
(flip 42)
;=> -42
```

Takes 'else' branch

```
(flip ‘‘42’’)
;=> -42
```

Local Reasoning



```
(defn flip [“42”]
  (if (string? “42”)
    (- (Long/parseLong “42”))
    (- n)))

(flip 42)
;=> -42

(flip “42”)
;=> -42
```

Takes ‘then’ branch

Java interop

Java

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

Test for nil (nil == null)

```
(upper-case nil)
;=> nil

(upper-case “abc”)
;=> “ABC”
```

Java interop

Java

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

(upper-case nil)
;=> nil

(upper-case “abc”)
;=> “ABC”
```

Java instance method call

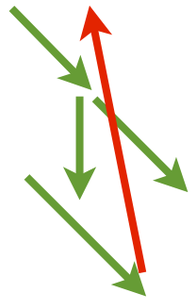
Java interop

Java

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

(upper-case nil)
;=> nil

(upper-case “abc”)
;=> “ABC”
```

Never nil, avoids NPE

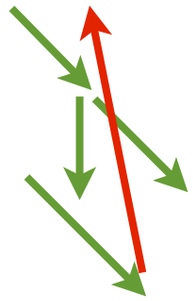
Multimethods



```
(defmulti flip (fn [n] (class n)))

(defmethod flip String [n]
  (- (Long/parseLong n)))

(defmethod flip Number [n]
  (- n))
```

Create new multimethod

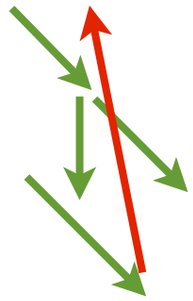
Multimethods



```
(defmulti flip (fn [n] (class n)))

(defmethod flip String [n]
  (- (Long/parseLong n)))

(defmethod flip Number [n]
  (- n))
```

Dispatch function

Multimethods



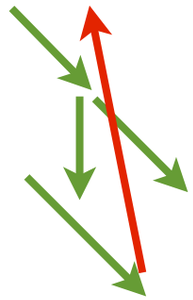
```
(defmulti flip (fn [n] (class n)))
```

Install method

```
(defmethod flip String [n]
  (- (Long/parseLong n)))
```

```
(defmethod flip Number [n]
  (- n))
```

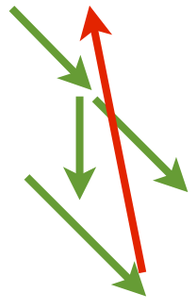
Multimethods



```
(defmulti flip (fn [n] (class n)))

(defmethod flip String [n]
  (- (Long/parseLong n)))

(defmethod flip Number [n]
  (- n))
```

Dispatch value

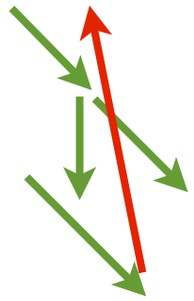
Multimethods



```
(defmulti flip (fn [n] (class n)))

(defmethod flip String [n]
  (- (Long/parseLong n)))

(defmethod flip Number [n]
  (- n))
```

Parameters

Multimethods

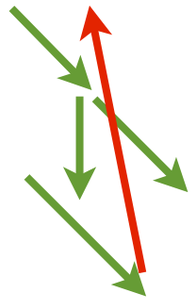


```
(defmulti flip (fn [n] (class n)))

(defmethod flip String [n]
  (- (Long/parseLong n)))
```

Method body

```
(defmethod flip Number [n]
  (- n))
```

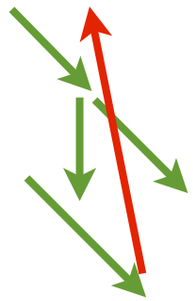
Multimethods



```
(defmulti flip (fn [“4”] (class “4”)))

(defmethod flip String [n]
  (- (Long/parseLong n)))

(defmethod flip Number [n]
  (- n))

(flip “4”)
```

1. Run dispatch function

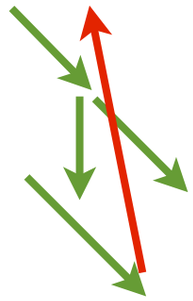
Multimethods



```
(defmulti flip (fn [“4”] (class “4”)))

(defmethod flip String [n]
  (- (Long/parseLong n)))

(defmethod flip Number [n]
  (- n))

(flip “4”)
```

Class literal
i.e. String.class

Multimethods

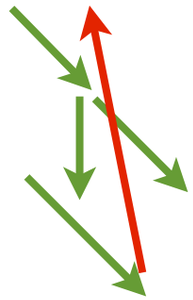


```
(defmulti flip (fn [“4”] (class “4”)))

(defmethod flip String [n]
  (- (Long/parseLong n)))

(defmethod flip Number [n]
  (- n))

(flip “4”)
```

```
(isa? (class “4”) String)
```

1. Run dispatch function
2. Use isa? to choose method

Multimethods



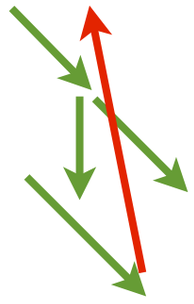
```
(defmulti flip (fn [“4”] (class “4”)))

(defmethod flip String [n]
  (- (Long/parseLong n)))

(defmethod flip Number [n]
  (- n))

(flip “4”)
```

```
(isa? (class “4”) String)
;=> true
```

Pick this method

1. Run dispatch function
2. Use isa? to choose method

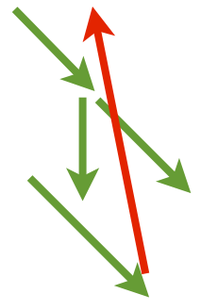
Multimethods



```
(defmulti flip (fn [“4”] (class “4”)))

(defmethod flip String [“4”]
  (- (Long/parseLong “4”)))

(defmethod flip Number [n]
  (- n))

(flip “4”)
```

1. Run dispatch function
2. Use isa? to choose method
3. Dispatch and return result

Multimethods



```
(defmulti flip (fn [“4”] (class “4”)))

(defmethod flip String [“4”]
  -4)

(defmethod flip Number [n]
  (- n))

(flip “4”)
;=> -4
```

1. Run dispatch function
2. Use isa? to choose method
3. Dispatch and return result

Understanding Clojure via Types

Understanding Clojure

Goal

Understand Clojure better

Method

?

Result

Insights about Clojure

Understanding Clojure

Goal

Understand
Clojure better

Method

Build and evaluate
a type system for Clojure

Result

Insights about
Clojure

clojure / core.typed

An optional type system for Clojure

2,509 commits **19** branches **139** releases **18** contributors

Branch: **master** New pull request New file Upload files Find file SSH git@github.com:clojure/core. Download ZIP

frenchy64 typo — Latest commit 730482e on Feb 21

images	typed clojure image	3 years ago
module-check	CTYP-308: enable sanity-checking compilation for core.typed-rt project	2 months ago
module-rt	CTYP-308: enable sanity-checking compilation for core.typed-rt project	2 months ago
.ackrc	Change for>, doseq>, fn>, loop> syntax, add deprecation notices.	3 years ago
.gitignore	CTYP-286: Revert to regular jar dependencies + bump t.a.j	5 months ago
.travis.yml	update travis configuration	8 months ago
CHANGELOG.md	0.3.22 release notes	2 months ago
CODE_OF_CONDUCT.md	update code of conduct	10 months ago
CONTRIBUTING.md	add code of conduct	10 months ago

Clojure

\+

Type

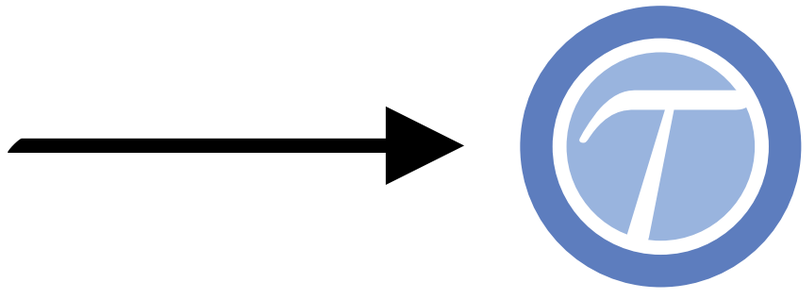
system

=

Typed Clojure

Building optional type systems

Clojure

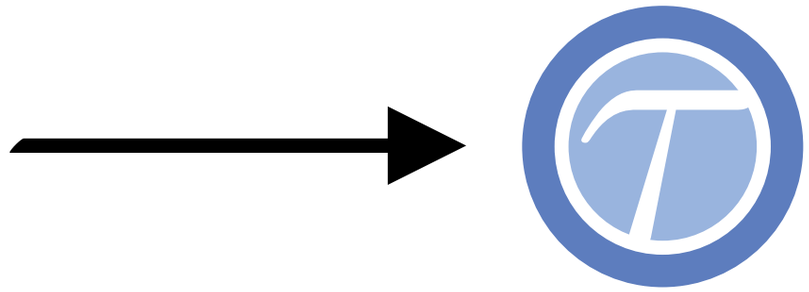


Typed Clojure

Work

1. Identify idiom

Clojure



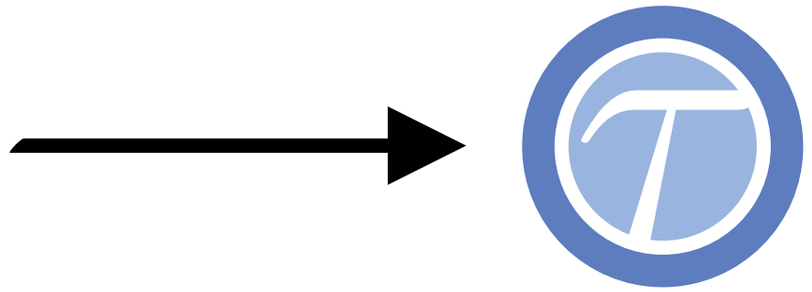
Typed Clojure

Work

1. Identify idiom

2. Extend type system

Clojure



Typed Clojure

Work

1. Identify idiom

2. Extend type system

1. Identify idiom

2. Extend type system

1. Identify idiom

2. Extend type system

⋮

⋮

Known keyword entries

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

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

(inc (:x (point 1 2)))
;=> 2
```

1. Identify idiom

Known keyword entries

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

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

(inc (:x (point 1 2)))
;=> 2
```

1. Identify idiom

Known keyword entries

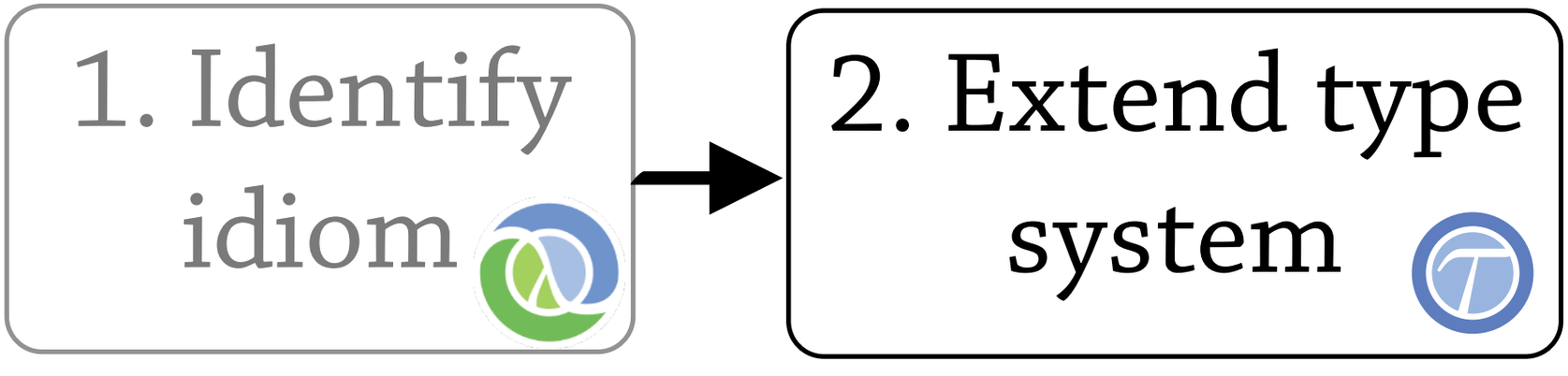
```
(ann point [Int Int -> ???])
(defn point [x y]
  {:x x
   :y y})

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

(inc (:x (point 1 2)))
;=> 2
```

1. Identify idiom

2. Extend type system

Known keyword entries

```
(ann point [Int Int -> '{:x Int :y Int}])
(defn point [x y]
  {:x x
   :y y})

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

(inc (:x (point 1 2)))
;=> 2
```



Known keyword entries

```
(ann point [Int Int -> ‘{:x Int :y Int}])
(defn point [x y]
  {:x x
   :y y})

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

(inc (:x (point 1 2)))
;=> 2
```

Evaluation

64% of HMap lookups resolve to ***known*** entries

1. Identify idiom 

2. Extend type system 

3. Evaluate ✓ 

Type-based control flow

```
(defn flip [n]
  (if (string? n)
    (- (Long/parseLong n))
    (- n)))
```

1. Identify idiom

Type-based control flow

```
(ann flip [(U Str Int) -> Int])
(defn flip [n]
  (if (string? n)
    (- (Long/parseLong n))
    (- n)))
```

Str

Int

Same name
but different type

1. Identify idiom

2. Extend type system

Type-based control flow

```
(ann flip [(U Str Int) -> Int])
(defn flip [n]
  (if (string? n)
    (- (Long/parseLong n))
    (- n)))
```

Str✓

Int✓

Solution:
Implement occurrence typing[1]

[1] Tobin-Hochstadt and Felleisen (ICFP '10)

1. Identify idiom

→

2. Extend type system

Preventing null-pointer exceptions

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

1. Identify idiom

Preventing null-pointer exceptions

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

Explicit nil type

1. Identify idiom

→

2. Extend type system

Preventing null-pointer exceptions

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

Nonnilable references

1. Identify idiom

2. Extend type system

Preventing null-pointer exceptions

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

Equivalent to `String` *in Java*

1. Identify idiom 

→

2. Extend type system 

Preventing null-pointer exceptions

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

Encode nil as a ***false*** *value*

1. Identify idiom

2. Extend type system

Preventing null-pointer exceptions

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

1. Identify idiom 

→

2. Extend type system 

Preventing null-pointer exceptions

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

Evaluation

62/62 methods avoid null-pointer exceptions

1. Identify idiom

2. Extend type system

3. Evaluate ✓

Multimethod control flow

```
(defmulti flip (fn [n] (class n)))

(defmethod flip String [n]
  (- (Long/parseLong n)))

(defmethod flip Number [n]
  (- n))

(flip-point n)
;=> -4
```

1. Identify idiom

Multimethod control flow

```
(ann flip [(U Str Int) -> Int])
(defmulti flip (fn [n] (class n)))

(defmethod flip String [n]
  (- (Long/parseLong n)))
```

Str

```
(defmethod flip Number [n]
  (- n))
```

Int

1. Identify idiom 

2. Extend type system 

Multimethod control flow

```
(ann flip [(U Str Int) -> Int])
(defmulti flip (fn [n] (class n)))

(defmethod flip String [n]
  (- (Long/parseLong n)))

(defmethod flip Number [n]
  (- n))
```

Str ✓

Int

Assume:

```
(isa? (class n) String)
```

Solution:
Assume dispatch when checking methods

1. Identify idiom

2. Extend type system

Multimethod control flow

```
(ann flip [(U Str Int) -> Int])
(defmulti flip (fn [n] (class n)))

(defmethod flip String [n]
  (- (Long/parseLong n)))

(defmethod flip Number [n]
  (- n))
```

Str ✓

Int ✓

Assume:

`(isa? (class n) String)`

Assume:

`(isa? (class n) Number)`

Solution:

Assume dispatch when checking methods

1. Identify idiom

2. Extend type system

Multimethod control flow

```
(ann flip [(U Str Int) -> Int])
(defmulti flip (fn [n] (class n)))

(defmethod flip String [n]
  (- (Long/parseLong n)))

(defmethod flip Number [n]
  (- n))
```

Evaluation

11 defmulti's,
89 defmethod's

1. Identify idiom

2. Extend type system

3. Evaluate ✓

Formalism

Occurrence typing overview

```
n:(U Str Int) ⊢ (if (string? n)
                    (- (Long/parseLong n))
                    (- n))
                  : Int
```

Under environment where
`n` is of type `(U Str Int)`

```
n:(U Str Int) ⊢ (if (string? n)
                    (- (Long/parseLong n))
                    (- n))
                : Int
```

This expression has type `Int`

```
n:(U Str Int) ⊢ (if (string? n)
                    (- (Long/parseLong n))
                    (- n))
                : Int
```

Check condition

```
n:(U Str Int) ⊢ (string? n) : Any
```

```
n:(U Str Int) ⊢ (if (string? n)
                    (- (Long/parseLong n))
                    (- n))
                : Int
```

```
n:(U Str Int) ⊢ (string? n) : Any
```

Problem!

n should be a Str

```
n:(U Str Int) ⊢ (- (Long/parseLong n)) : Int
```

```
n:(U Str Int) ⊢ (if (string? n)
                    (- (Long/parseLong n))
                    (- n))
                  : Int
```

Extend judgment with what we know if expression is true

```
n:(U Str Int) ⊢ (string? n) : Any ; n:Str
```

```
n:(U Str Int) ⊢ (- (Long/parseLong n)) : Int
```

```
n:(U Str Int) ⊢ (if (string? n)
                    (- (Long/parseLong n))
                    (- n))
                  : Int
```

Assume in branch!

```
n:(U Str Int) ⊢ (string? n) : Any ; n:Str
```

```
n:(U Str Int), n:Str ⊢ (- (Long/parseLong n)) : Int
```

```
n:(U Str Int) ⊢ (if (string? n)
                    (- (Long/parseLong n))
                    (- n))
                : Int
```

Same for 'else' branch

```
n:(U Str Int) ⊢ (string? n) : Any ; n:Str | n:(Not Str)
```

```
n:(U Str Int), n:Str ⊢ (- (Long/parseLong n)) : Int
n:(U Str Int), n:(Not Str) ⊢ (- n) : Int
```

```
n:(U Str Int) ⊢ (if (string? n)
                    (- (Long/parseLong n))
                    (- n))
                  : Int
```

Recap: Judgment extensions

```
n:(U Str Int) ⊢ (string? n) : Any ; n:Str | n:(Not Str)
```

Proposition environment

List of currently true propositions

Recap: Judgment extensions

```
n:(U Str Int) ⊢ (string? n) : Any ; n:Str | n:(Not Str)
```

‘Then’ proposition

True logical statement

when `(string? n)`

evaluates to a ***true*** *value*

Recap: Judgment extensions

```
n:(U Str Int) ⊢ (string? n) : Any ; n:Str | n:(Not Str)
```

‘Else’ proposition

True logical statement

when `(string? n)`

evaluates to a ***false*** *value*

```
⊢ (defmulti flip (fn [n] (class n)))
: (Multi [(U Int Str) -> Int]
         [x : (U Int Str) -> (U nil Class) ; … ; (class x)])
```

New work:
Multimethods + occurrence typing

```
⊢ (defmethod flip Number [n] (- n)) : …
```

Symbolic representation of dispatch function's return value

```
n:(U Str Int) ⊢ (class n) : (U nil Class) ; … ; (class n)
```

```
⊢ (defmulti flip (fn [n] (class n)))
: (Multi [(U Int Str) -> Int]
         [n : (U Int Str) -> (U nil Class) ; … ; (class n)])
```

```
⊢ (defmethod flip Number [n] (- n)) : …
```

```
n:(U Str Int) ⊢ (class n) : (U nil Class) ; … ; (class n)
```

```
⊢ (defmulti flip (fn [n] (class n)))
: (Multi [(U Int Str) -> Int]
         [n : (U Int Str) -> (U nil Class) ; … ; (class n)])
```

Remember in multimethod's type

```
⊢ (defmethod flip Number [n] (- n)) : …
```

```
n:(U Str Int) ⊢ (class n) : (U nil Class) ; … ; (class n)
```

```
⊢ (defmulti flip (fn [n] (class n)))
: (Multi [(U Int Str) -> Int]
         [n : (U Int Str) -> (U nil Class) ; … ; (class n)])
```

Check dispatch value

```
⊢ Number : (Value Number)
```

```
⊢ (defmethod flip Number [n] (- n)) : …
```

```
n:(U Str Int) ⊢ (class n) : (U nil Class) ; … ; (class n)
```

```
⊢ (defmulti flip (fn [n] (class n)))
: (Multi [(U Int Str) -> Int]
         [n : (U Int Str) -> (U nil Class) ; … ; (class n)])
```

Check method body

```
⊢ Number : (Value Number)        n:(U Str Int)  ⊢ (- n) : Int
```

```
⊢ (defmethod flip Number [n] (- n)) : …
```

```
n:(U Str Int) ⊢ (class n) : (U nil Class) ; … ; (class n)
```

```
⊢ (defmulti flip (fn [n] (class n)))
: (Multi [(U Int Str) -> Int]
         [n : (U Int Str) -> (U nil Class) ; … ; (class n)])
```

Assume most general type for parameter

```
⊢ Number : (Value Number)        n:(U Str Int) ⊢ (- n) : Int
```

```
⊢ (defmethod flip Number [n] (- n)) : …
```

```
n:(U Str Int) ⊢ (class n) : (U nil Class) ; … ; (class n)
```

```
⊢ (defmulti flip (fn [n] (class n)))
: (Multi [(U Int Str) -> Int]
         [n : (U Int Str) -> (U nil Class) ; … ; (class n)])
```

Problem!

n should be Int

```
⊢ Number : (Value Number)        n:(U Str Int) ⊢ (- n) : Int
```

```
⊢ (defmethod flip Number [n] (- n)) : …
```

```
n:(U Str Int) ⊢ (class n) : (U nil Class) ; … ; (class n)
```

```
⊢ (defmulti flip (fn [n] (class n)))
: (Multi [(U Int Str) -> Int]
         [n : (U Int Str) -> (U nil Class) ; … ; (class n)])
```

n:Num

Solution

Reconstruct dispatch invariants

```
⊢ Number : (Value Number)        n:(U Str Int), n:Num ⊢ (- n) : Int
```

✓

```
⊢ (defmethod flip Number [n] (- n)) : …
```

Evaluation

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

Avoiding null-pointer exceptions

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

Insight:

Clojure programmers use simple local reasoning to avoid null-pointer exceptions

Absence of map entries

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

Types must track *absence* of :x entry to prevent bad type

Absence of map entries

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

Fully specified HMaps:

```
(HMap :complete? true
      :mandatory {:y y})
```

Absence of map entries

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

Evaluation
27% of keyword lookups on HMaps either ***optional*** or ***absent***

Insight:

Clojure programmers reason about
*the **presence** and **absence** of specific map keys*

Good idioms?

- ✓ Soundness proof
- ✓ Straightforward design

Real idioms?

- ✓ Popular implementation
- ✓ Evaluation

Clojure encourages disciplined programming

Typed Clojure is a real tool used today

Typed Clojure is a real tool used today

Ambrose Bonnaire-Sergeant

@ambrosebs

https://github.com/clojure/core.typed

Thanks!