# Qualifying Exam

Ambrose Bonnaire-Sergeant (0003410123)

January 25, 2018

## Contents

## 1 Question 1

Analyze the space and time complexity of your approach to dynamic tracing & subsequent type inference for Typed Clojure. Are you able to bound space use at all by reducing traces as they are collected? Please analyze a related system, Daikon (https://plse.cs.washington.edu/daikon/), along the same lines. How expressive are Daikons invariants compared to yours? How much space and runtime overhead does it impose? If Daikon's inferred invariants were to become part of a (refinement) type system, how powerful would it need to be?

## 1.1 Space complexity of dynamic tracing

## 1.2 Time complexity of dynamic tracing

To avoid the full cost of naive dynamic tracing, we leverage several known techniques from the higher-order contract checking literature.

Firstly, most tracked Clojure collections are traversed lazily as they are used. This includes (potentially infinite) lazy sequences and hash maps. Vectors are not currently lazily traversed, and we have not tried any vector-heavy benchmarks to observe the performance effect (but we hypothesize matrix-heavy code would suffer from a significant tracking penalty).

Functions are similarly wrapped a la higher-order contract checking and tracked when invoked. Each function call must traverse its argument list to track them.

Each function return tracks its return value.

Objects simply record their class, including Arrays.

Map wrappers are space-efficient with respect to the stack, and redundant wrappers collapse so there is only ever one level of wrapping.

Increases the more values are reachable

## 1.3 Space complexity of type inference

## 1.4 Time complexity of type inference

Reference:

**I** number of collected inference results

**U** maximum number of union members (unordered types)

**D** maximum depth of types

**W** maximum width of non-union types (ordered types) (eg. HMap entries, function positions)

**A** maximum number of aliases in alias environment (reachable from the type environment)

### 1.4.1 Join

```
join(D, W, U) = O(D * max(W, U^2))
```

Joining two union types involves joining all the combinations of the union members ($U^2$ joins).

Joining two non-union types that are not the same sort of type is constant.

Joining two non-union types that are the same sort of type joins each of the members of its types pairwise, of which the maximum number is $W$.

A maximum number of $D$ recursive joins can occur, and $max(W, U^2)$ work is done at each level, so time complexity is $O(D * max(W, U^2))$.

### 1.4.2 Naive type environment creation

1. Build naive type environment. Folds over inference results, 'update' from the top of each type. Naive algorithm traverses depth/width of type

```
naive(I, D, W, U) = O(I * (D + join(D, W, U)))
```

Iterate over each inference result, there are $I$ of them.

Build up a sparse type from the inference result, maximum depth $D$.

Join this type with the existing type taking $join(D, W, U)$.

Since we do $(D + join(D, W, U))$ work for each result, time complexity is $O(I * (D + join(D, W, U)))$.

### 1.4.3 Naive type environment creation (optimized)

1. Build naive type environment (optimized) Group inference results by path prefixes. Then join the groups from the longest prefixes first, and use previous results to avoid recalulating joins.

$$optimized\Gamma(I, D, W, U) = O(I * (D + join(D, W, U)))$$

### 1.4.4 Squash Vertically

Iterate down each type in the type environment and merge similarly "tagged" maps, resulting in a possibly recursive type.

Each iteration involves:

- *alias-hmap-type* Ensure each HMap in the current type corresponds to an alias.
  - Involves walking the current type once
  - $alias\_hmap\_type(D, W, U) = O(D * W * U)$
- *squash-all* Each alias mentioned in the current type is "squashed".
  - At worst, could join each alias together.
  - $squash\_all(A, D, W, U) = O(A * join(D, W, U))$

$$squash\_vertically(\Gamma, A, D, W, U) = O(|\Gamma| * (D * W * U + A * join(D, W, U)))$$

### 1.4.5 Squash Horizontally

Iterate over the reachable aliases (from the type environment) several times, merging based on several criteria.

These are the passes:

- *group-req-keys* Merge HMap aliases with similar keysets, but don't move tagged maps
  - First groups aliases into groups indexed by their keysets, then joins those groups together, merging each group into its own alias.
  - $group\_req\_keys(A, D, W, U) = O(A + A * join(D * W * U))$
- *group-likely-tag* Merge HMap aliases with the same tag key/value pair
  - First groups aliases into groups indexed by their likely tag key/value, then joins those groups together, merging each group into its own alias.
  - $group\_likely\_tag(A, D, W, U) = O(A + A * join(D * W * U))$
- *group-likely-tag-key* Merge HMap aliases with the same tag key. This gathers HMaps with the same tag key into the large unions seen in the algorithm's final output.
  - First groups aliases into groups indexed by their likely tag key, then joins those groups together, merging each group into its own alias.
  - $group\_likely\_tag\_key(A, D, W, U) = O(A + A * join(D * W * U))$

$$\begin{aligned} squash\_horizontally(\Gamma, A, D, W, U) = O(&group\_req\_keys(A, D, W, U) \\ &+ group\_likely\_tag(A, D, W, U) \\ &+ group\_likely\_tag\_key(A, D, W, U)) \end{aligned}$$

### 1.4.6 Overall time complexity

The overall time complexity is the sum of the previous passes.

$$\begin{aligned} alg(\Gamma, A, I, D, W, U) = O(&optimized\Gamma(I, D, W, U) \\ &+ squash\_vertically(\Gamma, A, D, W, U) \\ &+ squash\_horizontally(\Gamma, A, D, W, U)) \end{aligned}$$

Based on this analysis, we have a few observations:

- the larger the number of aliases, the slower the algorithm
  - both the phases that merge aliases traverse and potentially the aliases in the type environment together.
- the larger the size of unions, the slower the algorithm
  - *join* is quadratic in the size of the largest union.

## 1.5 Benchmarks

To test the time complexity of the implementation of the type reconstruction algorithm, we devise several benchmarks.

The benchmarks are designed to emperically determine the influence of the size of the largest union type on the time complexity of the type reconstruction algorithm.

### 1.5.1 Benchmark 1: 2 tags

The first benchmark generates deep inputs using one of 2 "tagged" maps. This is equivalent to seeding the algorithm with a large Lisp-style list with cons/null constructors.

Since there is only one interesting tagged map, the early phases of the algorithm should collapse (or "squash") these repeated tag occurrences into a small (recursive) union, thus minimizing the size of the largest union in the later stages of the algorithm.

Benchmark 1 takes as input the number of "cons" tagged maps to wrap around a "null" tag. For example, input 5 starts the algorithm with the type

```
'{:tag ':cons,
  :cdr '{:tag ':cons,
          :cdr '{:tag ':cons,
                :cdr '{:tag ':cons,
                       :cdr '{:tag ':cons,
                              :cdr '{:tag ':null}}}}}}
```

The final output of the algorithm is a recursive type with 2 tags.

```
(defalias Tag (U '{:tag ':cons, :cdr Tag} '{:tag ':null}))
```

All inputs to Benchmark 1 greater than 0 result in this same calculated type.

### 1.5.2 Benchmark 2: Many tags

The second benchmark forces the largest union to be linear to the depth of the input type. It achieves this with a similar approach to Benchmark 1, except every level of the initial type is tagged with a unique type.

Since the recursive type reconstruction only collapses identically-tagged maps, we preserve a large union throughout the algorith.

We demonstrate the behavior of Benchmark 2 at depth 5. This is the initial type we use to recover types from:

```
'{:tag ':cons5,
  :cdr '{:tag ':cons4,
         :cdr '{:tag ':cons3,
                :cdr '{:tag ':cons2,
                       :cdr '{:tag ':cons1,
                              :cdr '{:tag ':null}}}}}}
```

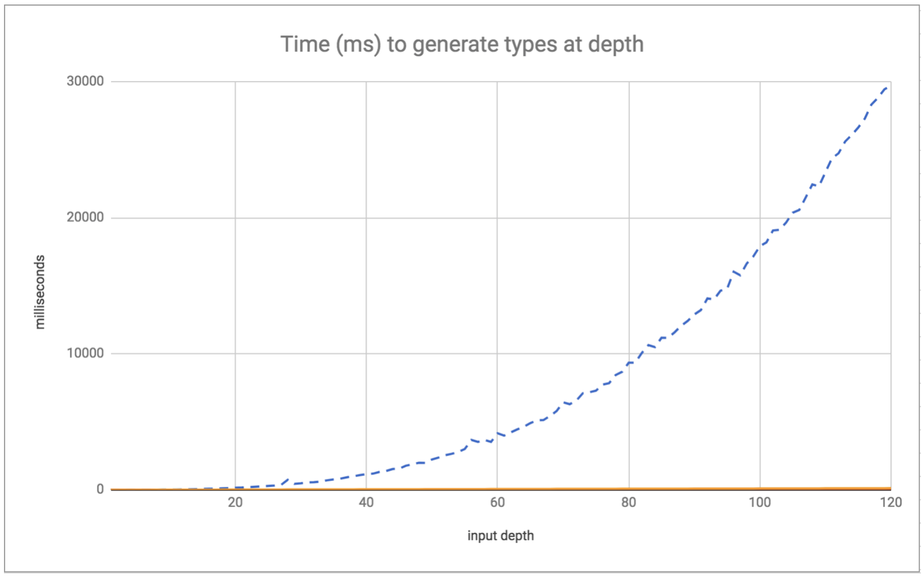
And, since the input has 6 unique tags, our final recursive type has a recursive union of width 6.

```
(defalias
  Tag
  (U
    '{:tag ':cons1, :cdr Tag}
    '{:tag ':cons2, :cdr Tag}
    '{:tag ':cons3, :cdr Tag}
    '{:tag ':cons4, :cdr Tag}
    '{:tag ':cons5, :cdr Tag}
    '{:tag ':null}))
```

The width of the reconstructed type is linear in the depth of the input type. For example, Benchmark 2 with depth 10 results in:

```
(defalias
  Tag
  (U
    '{:tag ':cons1, :cdr Tag}
    '{:tag ':cons2, :cdr Tag}
    '{:tag ':cons3, :cdr Tag}
    '{:tag ':cons4, :cdr Tag}
    '{:tag ':cons5, :cdr Tag}
    '{:tag ':cons6, :cdr Tag}
    '{:tag ':cons7, :cdr Tag}
    '{:tag ':cons8, :cdr Tag}
    '{:tag ':cons9, :cdr Tag}
    '{:tag ':cons10, :cdr Tag}
    '{:tag ':null}))
```

### 1.5.3 Results



Benchmark 1 is represented by the red solid line, benchmark 2 by the blue dotted line.

The graph shows the performance of Benchmark 1 is constant, because the width of the largest union is always 2 in an early phase of the reconstruction algorithm.

The results for Benchmark 2 show the reconstruction algorithm is quadratic with respect to the size of the largest union.

These observations are consistent with the prior theoretical analysis.

## 1.6 Can space use be bounded by reducing traces as collected?

Traces in Typed Clojure's dynamic analysis are accumulated online, and then folded into a type environment offline. However, this fold operation is commutative with respect to the order of traces, so performing this fold online would eliminate the need to store traces in memory.

Space is reduced further, then, by leveraging `join` to eagerly simplify the accumulated type environment. For example, heterogeneous maps with similar keysets could merged, possibily using optional key entries, saving space by preventing very large redundant unions.

It is unclear if it is possible to perform more sophisticated analyses online, in particular the recursive type reconstruction algorithm. Since the resulting annotations are very compressed compared to intermediate points in the analysis, fully or partially performing this analysis online may drastically decrease space usage where recursively defined maps are used, and very deep examples are found.

## 1.7 Daikon's expressivity vs Typed Clojure's dynamic inference

Processing in Clojure is done via functions, often with immutable variables and collections. Invariants

The Java language requires type annotations for every variable, which Daikon utilizes. This also means basic type information does not need to be collected about variables, past whether they are null.

Daikon is interested in invariants between method entry and method exit. For example, how a mutable variable might evolve over the course of a method call, or over the course of an object's life. This kind of data is less interesting in Clojure, since mutability, especially the usage of unsynchronized mutable local variable, is discouraged and usually seen as for experts only.

## 1.8 Space/time overhead of Daikon's dynamic tracing

At each method entry/exit point, record the value of all variables in scope.

How to track values:

For each Java primitive, record its value.

For each Array, traverse its contents and collect hash codes and/or primitive values.

Otherwise, get the class of the current object.

## 1.9 Space/time overhead of Daikon's type inference

## 1.10 How to type check Daikon's invariants

"Simplify" is a theorem prover for Java. Daikon can compile its invariants to Simplify.

Simplify implements the following theories.

1. The theory of equality, `=`

2. The theory of arithmetic with functions `+`, `*`, `-`, and relation symbols `>`, `<`, `<=`, and `>=`.

3. The theory of maps with two functions `select` and `store` (ie. get/set), and two additional axoims.

4. Partial orders (?)

These could be encoded in Dependent Typed Racket, since it supports propositions in linear arithmetic constraints about variables, pairs, car, and cdr.

## 2 Question 2

Examine the use of Clojure's core.spec contract system in several real world code bases. Look at what features are used, and how precise specifications are. Analyze how specifications address the lack of higher-order contracts by looking at the frequency of higher-order function contracts vs higher-order functions that omit specifications of higher-order arguments or results.

### 2.1 What spec features are used in real systems

### 2.2 How precise are spec annotations in practice?

### 2.3 How frequently are higher-order functions annotated with higher-order specs? Why?

$$\begin{array}{lll}
d, e & ::= x \mid v \mid (e\ e) \mid \lambda x^{\tau}.e \mid (\text{if } e\ e\ e) \mid (\text{do } e\ e) & \text{Expressions} \\
v & ::= l \mid \{\} \mid n \mid m \mid [\rho, \lambda x^{\tau}.e]_{\mathsf{c}} & \text{Values} \\
m & ::= \{\overrightarrow{v \mapsto v}\} & \text{Map Values} \\
G & ::= (\text{get } e\ e) \mid (\text{assoc } e\ e\ e) &
\end{array}$$

Figure 1: Syntax of Terms and Specs

## 3 Question 3

Write a formal model of Clojure with core.spec, and implement it in PLT Redex. Formulate a consistency property between contracted and uncontracted execution, and test it in redex.