Qualifying Exam

Ambrose Bonnaire-Sergeant (0003410123)

February 21, 2018

Contents

1 Question 1 **3**
- 1.1 Space complexity of dynamic tracing . . . 4
- 1.2 Benchmarks . . . 4
- 1.3 Time complexity of dynamic tracing . . . 4
- 1.4 Space complexity of type inference . . . 5
- 1.5 Time complexity of type inference . . . 5
  - 1.5.1 Join . . . 5
  - 1.5.2 Naive type environment creation . . . 5
  - 1.5.3 Naive type environment creation (optimized) . . . 6
  - 1.5.4 Squash Vertically . . . 6
  - 1.5.5 Squash Horizontally . . . 6
  - 1.5.6 Overall time complexity . . . 7
- 1.6 Benchmarks . . . 7
  - 1.6.1 Benchmark 1: 2 tags . . . 7
  - 1.6.2 Benchmark 2: Many tags . . . 8
  - 1.6.3 Results . . . 9
- 1.7 Can space use be bounded by reducing traces as collected? . . . 10
- 1.8 Daikon . . . 10
  - 1.8.1 Daikon's expressivity vs Typed Clojure's dynamic inference 11
  - 1.8.2 Space/time overhead of Daikon's dynamic tracing . . . 11
  - 1.8.3 Space/time overhead of Daikon's type inference . . . 12
  - 1.8.4 How to type check Daikon's invariants . . . 12

2 Question 2 **13**
- 2.1 What spec features are used in real systems . . . 13
- 2.2 How precise are spec annotations in practice? . . . 13
- 2.3 How frequently are higher-order functions annotated with higher-order specs? Why? . . . 13

3 Question 3 **14**
3.1 Formal model 14
3.2 Consistency property 14
3.2.1 Counter-example 14
3.3 Redex model 14

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?

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

1.1 Space complexity of dynamic tracing

Map and function wrappers are space-efficient with respect to the stack, and redundant wrappers of the same type collapse so there is at most one level of wrapping of the same type at a given level.

Some structural sharing of tracking data helps reduce the memory footprint of the tracking results.

The heap size of the tracking data is linear in the number of observed tracking points.

1.2 Benchmarks

We use several small benchmarks to demonstrate the space-efficiency of function wrappers, and the linear nature of the tracking data.

1.3 Time complexity of dynamic tracing

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 return tracks its return value.

Non-Clojure 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.

1.4 Space complexity of type inference

The type inference algorithm uses a linear amount of memory with respect to the number of distinct HMap types in the input. At its largest space usage, the algorithm creates a new alias for each HMap object.

However, aliases consist of a symbol and an entry in a hash-map, so the constant factors are small.

1.5 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)

1.5.1 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 $O(D * max(W, U^2))$.

1.5.2 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 $O(I * (D + join(D, W, U)))$.

1.5.3 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.

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

1.5.4 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.
  - $squash\_all(A, D, W, U) = O(A * join(D, W, U))$

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

1.5.5 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.
  - $group\_likely\_tag\_key(A, D, W, U) = O(A + A * join(D * W * U))$

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

1.5.6 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.

1.6 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.

1.6.1 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.

1.6.2 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}}}}}}
```

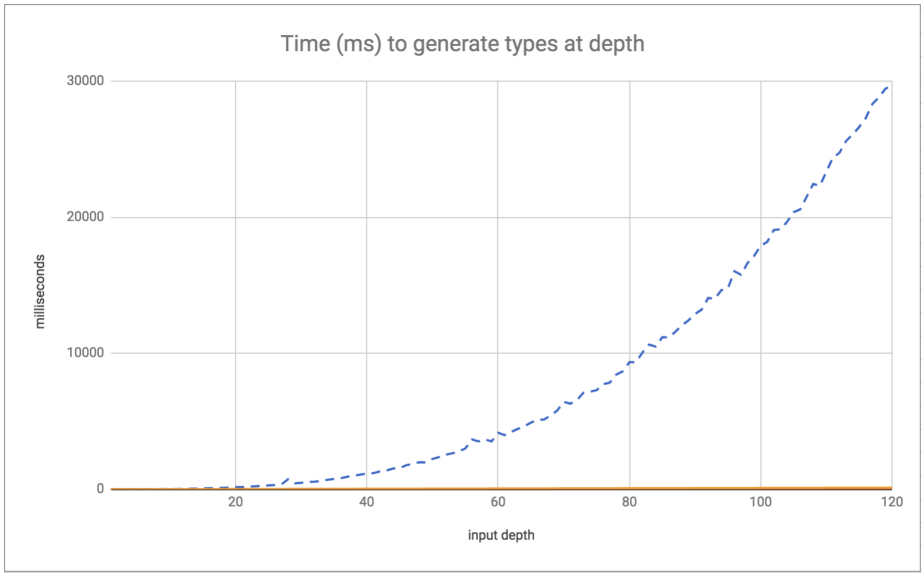
And, since the input has 6 unique tags, our final recursive type has a recursive union of width 6.

```
(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}))
```

1.6.3 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.

1.7 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.

1.8 Daikon

Daikon is a related system to Typed Clojure's dynamic type inference. However, Daikon is built for "invariant detection", while our system is designed for "dynamic type inference" or "value profiling".

There are two common implementation strategies for such tools. The first strategy, "ruling-out" (for invariant detection), assumes all invariants are true and then use runtime analysis results to rule out impossible invariants. The second "building-up" strategy (for dynamic type inference) assumes nothing and then uses runtime analysis results to build up invariant/type knowledge.

Both strategies have different space behavior with respect to representing the set of known invariants. The ruling-out strategy typically uses a lot of memory at the beginning, but then can free memory as it rules out invariants. For example, if `odd(x)` and `even(x)` are assumed, observing `x = 1` means we can delete and free the memory recording `even(x)`. Alternatively, the building-up strategy uses the least memory storing known invariants/types at the beginning, but increases memory usage as more the more samples are collected. For example, if we known `x : Any`, and we observe `x = "a"` and `x = 1` at different points in the program, we must use more memory to store the union `x : String` $\cup$ `Integer` in our set of known invariants.

Examples of invariant detection tools include Daikon [1], DIDUCE [2], and Carrot [6], and typically enhance statically typed languages with more expressive types or contracts. Examples of dynamic type inference include Rubydust [3], JSTrace [7], and TypeDevil [5], and typically target untyped languages.

There is some overlap between invariant detection and dynamic type inference tools. Usually, invariant detection detects very expressive relationships between program variables; for example, for array `a` and index `i` variables, a derived invariant might be `0 < a[i]`. On the other hand, dynamic type in-

ference (or value profiling) often just records basic nominal or structural type information—it is generally applied to untyped languages where basic static type information is absent.

1.8.1 Daikon's expressivity vs Typed Clojure's dynamic inference

Daikon can reason about very expressive relationships between variables using properties like ordering ($x < y$), linear relationships ($y = ax + b$), and containment ($x \in y$). It also supports reasoning with "derived variables" like fields ($x.f$), and array accesses ($a[i]$).

Typed Clojure's dynamic inference can record heterogeneous data structures like vectors and hash-maps, but otherwise cannot express relationships between variables.

There are several reasons for this. The most prominent is that Daikon primarily targets Java-like languages, so inferring simple type information would be redundant with the explicit typing disciplines of these languages. On the other hand, the process of moving from Clojure to Typed Clojure mostly involves writing simple type signatures without dependencies between variables. Typed Clojure recovers relevant dependent information via occurrence typing, and gives the option to manually annotate necessary dependencies in function signatures when needed.

1.8.2 Space/time overhead of Daikon's dynamic tracing

Performance of dynamic tracing is not directly addressed in the Daikon literature, who only provide complexity analyses and optimizations for storing and checking invariants *after* samples have been collected.

I have manually examined Chicory (the Java front-end to Daikon) to see how dynamic tracing is implemented. I found that Daikon records the value of all variables in scope at each method entry/exit point [1]. Then, to record values in Daikon, the following algorithm is used:

- If the value is a Java primitive, record its value.
- If the value is an array, traverse its contents and record identity hash codes and/or primitive values.
- Otherwise, record the class of the current object.

Notice this algorithm is non-recursive—while arrays are traversed eagerly, they are only traversed one level via an identity hash code summary (the closest equivalent to pointer addresses on the JVM). This is significantly different to Typed Clojure's value tracing algorithm, which recursively (but lazily) traverses potentially-deep data structures.

Another difference is that Typed Clojure's dynamic tracing only tracks values for arguments/returns of a function, and ignores any variables that in are

[1] Implemented in `daikon.chicory.DaikonVariableInfo`

scope. There are several reasons behind this decision. First, Java-like object-oriented languages use fields as implicit arguments to methods, and Daikon distinguishes method-level, and class-level invariants which is achieved by checking class-level invariants during method calls. In Clojure, methods are replaced with pure functions (their output is defined only by the explicitly passed arguments), so the method/class-level distinction is not applicable.

Second, Daikon chooses to reason about local mutation in Java-like languages, and so must record the values of the same variables different program points to observe mutation. However, local unsynchronized mutation is non-idiomatic in Clojure so re-tracking variables is almost always redundant—mutation is often via synchronized global variables that can be instrumented once-and-for-all.

1.8.3 Space/time overhead of Daikon's type inference

The overhead of likely invariant detection is described by Perkins and Ernst [4]. They analyze the overhead of storing and checking the set of invariants that are currently true. Their presentation includes a simple incremental algorithm that features no optimizations, then they propose several candidate optimizations and emperically compare the performance of each approach.

The simple incremental algorithm

1.8.4 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?

$$
\begin{array}{lll}
e & ::= x \mid v \mid (e\ e) \mid \lambda x.e \mid (\text{if } e\ e\ e) & \text{Expressions} \\
v & ::= \{\} \mid \mathsf{err} \mid n \mid m \mid [\rho, \lambda x.e] & \text{Values} \\
m & ::= \{\overrightarrow{v \mapsto v}\} & \text{Map Values} \\
\rho & ::= [\overrightarrow{x \mapsto v}] & \text{Environments} \\
p & ::= \text{zero?} \mid \text{number?} \mid \text{boolean?} & \text{Predicates} \\
c & ::= p \mid \text{get} \mid \text{assoc} & \text{Constants} \\
\mathcal{E} & ::= [] \mid (\mathcal{E}\ e) \mid (v\ \mathcal{E}) \mid (\text{if } \mathcal{E}\ e\ e) & \text{Contexts}
\end{array}
$$

Figure 1: Syntax of Terms in λc

$$
e ::= \ldots \mid \qquad \text{Expressions}
$$

Figure 2: Syntax of λc_s (extending λc)

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.`

3.1 Formal model

We devise a base formal model for Clojure called λc. We extend λc with clojure.spec without function contracts, and call this model λc_s. Then, we extend λc_s to support clojure.spec function contracts, and call this final model λc_s^f.

3.2 Consistency property

The base

3.2.1 Counter-example

3.3 Redex model

We model λc as a Redex language called *Clojure*. The model includes also recursive functions and nil.

To explore Redex, we use an Eval-Apply-Continue machine

$$
e ::= \ldots \mid \qquad \text{Expressions}
$$

Figure 3: Syntax of λc_s^f (extending λc_s)

References

[1] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants, 2006.

[2] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 291–301. IEEE, 2002.

[3] J. hoon (David) An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for ruby, 2010.

[4] J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *In Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004*, pages 23–32, 2004.

[5] M. Pradel, P. Schuh, and K. Sen. Typedevil: Dynamic type inconsistency analysis for javascript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 314–324. IEEE, 2015.

[6] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. *arXiv preprint cs/0310040*, 2003.

[7] C. Saftoiu. Jstrace: Run-time type discovery for javascript. Technical report, Technical Report CS-10-05, Brown University, 2010.