Qualifying Exam

Ambrose Bonnaire-Sergeant (0003410123)

January 5, 2018

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?
```

Space/time complexity of dynamic tracing

Tracked Clojure collections are traversed lazily as they are used.

Functions are 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

Space/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)

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

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

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

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.

Daikon's expressivity vs Typed Clojure's dynamic inference

A significant difference between Typed Clojure's dynamic inference and Daikon's is that the former targets (primarily) structural types, and the latter nominal types. In Clojure, the primary data is in the form of data structures, and in Java it is in the form of classes.

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.

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.

Space/time overhead of Daikon's type inference

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.

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
```

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

Figure 1: Syntax of Terms and Specs

`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.`

What spec features are used in real systems

How precise are spec annotations in practice?

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

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.`