Qualifying Exam

Ambrose Bonnaire-Sergeant (0003410123)

March 20, 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?`

We begin by analyzing our approach to dynamic tracing and type inference for Typed Clojure's dynamic type inference.

1.1 Dynamic tracing Performance

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

1.1.1 Space complexity of dynamic tracing

The space complexity of dynamic tracing in Typed Clojure is similar to proxy-based higher-order contract checking (so, reasonable, with some constant overhead).

Map and function wrappers are space-efficient with respect to the stack, with redundant wrapper of the same type collapsing to at most one level of wrapping.

Some structural sharing of tracking data helps reduce the memory footprint of the tracking results. Specifically, the original source of a value is stored in a "path" (similar to a blame label in contract systems), and appending to this path uses constant heap space since it is represented using a immutable vector with structural sharing.

1.1.2 Time complexity of dynamic tracing

Most tracked Clojure collections are traversed lazily as they are used, so incurs a $O(1)$ cost to each lookup. 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, so incur an overhead on every invocation. Non-Clojure Objects simply record their class (including arrays).

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

Figure 1: Reference for variables used in time complexity analysis

1.2 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.

The algorithm also can use more space as more samples are processed. For example, a large union type could be accumulated by combining several samples.

1.3 Time complexity of type inference

To analyze the time complexity of dynamic type inference (after gathering samples), we reference several variables defined in Figure 1. The variable `I` denotes the number of samples collected during dynamic tracing. The variable `U` denotes the maximum width of a union during the entire execution of type inference. The maximum depth and width of non-union types are denoted by `D` and `W` respectively. Finally, `A` references the number of (reachable) aliases in the alias environment.

The entire algorithm is split into several passes, many of which rely on the `join` function, which we analyze in Section 1.3.1.

First, a naive type environment is generated. Two approaches are analyzed, one slow (Section 1.3.2), and one fast (Section 1.3.3). The implementation uses the second version in practice.

Then several passes are applied to the resulting type and alias environments. First *squash_vertically* (Section 1.3.4) creates local recursive types. Then *squash_horizontally* (Section) merges similar recursive types that occur in different type/alias environment entries. The overall time complexity is summarized in Section 1.3.6.

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

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

1.3.3 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.

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

1.3.4 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.

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

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

1.3.5 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.
  - $group\_likely\_tag\_key(A, D, W, U) = O(A + A * join(D * W * U))$

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

1.3.6 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.

1.4 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.

1.4.1 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.

1.4.2 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}}}}}}
```

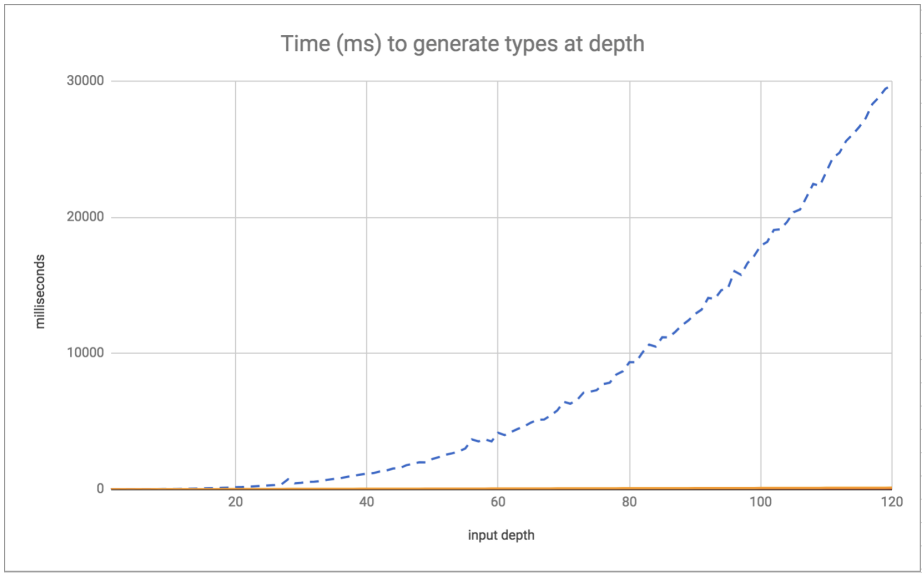
And, since the input has 6 unique tags, our final recursive type has a recursive union of width 6.

```
(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}))
```

1.4.3 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.

1.5 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.

1.6 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 [3], DIDUCE [4], and Carrot [8], and typically enhance statically typed languages with more expressive types or contracts. Examples of dynamic type inference include Rubydust [5], JSTrace [9], and TypeDevil [7], 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 inference (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.

1.6.1 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.

1.6.2 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 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.

1.6.3 Space/time overhead of Daikon's type inference

The overhead of likely invariant detection is described by Perkins and Ernst [6]. They analyze the overhead of storing and checking the set of invariants that are currently true. Their presentation includes a simple incremental algorithm that

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

features no optimizations, then they propose several candidate optimizations and emperically compare the performance of each approach.

The space complexity of the simple incremental algorithm is dominated by the size of the grammar of properties. For example, if the grammar of properties consists of $=$ and *even*, with 3 variables x, y, z, the initial (and largest) set of invariant assumptions is $x = y$, $y = z$, $x = z$, $even(x)$, $even(y)$, and $even(z)$. For reference, Daikon enables 152 properties by default, with 12 "derived variables", the latter of which provide properties on composite variables like $a[x] = a[z]$. After a static analysis pass ruling out nonsensical invariants, the space usage is at least $O(v^9)$, where v is the number of variables in scope.

The time complexity of *checking* invariants for each sample is similarly dominated by the size of the grammar of properties. They note, however, most invariants are removed quickly (after $O(1)$ samples), so performance improves as more samples are collected.

1.6.4 Checking Daikon's invariants in a refinement type system

Daikon has expressive invariants, but can they be statically verified? Yes, in fact Daikon supports generating annotations for a Java-based theorem prover called Simplify [2].

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 clojure.spec

Recently, Clojure added a runtime verification system to its core library called `clojure.spec`. It resembles common approaches to runtime verification, such as Racket's contract system, but is different in several important ways.

Firstly, `clojure.spec` is designed to treat most values as "data at rest". That is, at verification sites, values are eagerly traversed without waiting to see if or how the program actually uses them. When we consider that `clojure.spec` treats infinite streams and functions as data at rest, we begin to see the tradeoffs that have been made.

Secondly, specifications (called "specs") are not enforced by default. Users must opt-in to enforcing specs via an explicit instrumentation phase. This is also different than most contract systems, many of which are enforced by default. There is no standard way to integrate spec enforcement into a test suite, so it is difficult to tell whether specific specs are primarily unchecked documentation, or actually used for runtime verification.

Since `clojure.spec` has a unique feature set amongst runtime verification libraries, it is interesting to consider how programmers use `clojure.spec` in practice. For example, do programmers find the semantics of treating functions as data at rest useful?

Unfortunately since specs are opt-in, it is difficult to correlate someone writing a spec with that person *using* the spec, implying spec's semantics as being useful. Nevertheless, in the following sections we attempt to draw conclusions about spec's common usage based mostly on the frequency of spec annotations.

2.2 Function specifications in clojure.spec

From here, we map the namespace prefix `s` to `clojure.spec.alpha`, and `stest` to `clojure.spec.test.alpha`.

```
(require '[clojure.spec.alpha :as s])
(require '[clojure.spec.test.alpha :as stest])
```

There are two kinds of function checking semantics in `clojure.spec`. We use `intmap`, a higher-order function that maps a function over a collection of ints, to demonstrate both semantics.

```
(defn intmap
  "Maps a collection of ints over a function."
  [f c]
  (map f c))
```

If the programmer wants to write a higher-order function spec to verify `intmap`, they might write the following spec.

```
(s/fdef intmap
  :args (s/cat :f (s/fspec :args (cat :x int?) :ret int?)
               :c (s/coll-of int?))
  :ret (s/coll-of int?))
```

The `s/fdef` form signals we are annotating a top-level function, in this case `intmap`. Argument specs are provided with the `:args` keyword option in the form of the "tagged" heterogeneous collection spec `s/cat`—here 2 arguments are allowed, tagged as `:f` for the function and `:c` as the collection.

The `s/fspec` spec is another kind of function spec, specifically for non-top-level functions (such as function arguments to top-level functions). It has a similar syntax to `s/fdef`, but a function name is not provided.

In a nutshell, `s/fdef` provides traditional proxy-based verification semantics while `s/fspec` uses eager *generative testing* to exercise a function before letting it pass the spec boundary, bare (without a proxy).

We will now demonstrate how the following call gets checked.

```
(intmap inc [1 2 3])
;=> (2 3 4)
```

First, the programmer instruments `intmap` with:

```
(stest/instrument `intmap)
```

This mutates the top-level binding associated with `intmap`, wrapping a function proxy around the original value.

Now, when checking `(intmap inc [1 2 3])`, the `inc` function is called several hundred times with generated values conforming to `int?`, and checks each call returns an `int?`. Then, `[1 2 3]` is eagerly checked against `(s/coll-of int?)`. The original `intmap` function is then called with the original arguments, yielding a value `(1 2 3)`. Instrumentation does not check return value specs, so `(s/coll-of int?)` is ignored, and the original return value is passed to the calling context.

2.3 Research questions

I aim to answer these research questions:

- What does the frequency and applications of `fdef` and `fspec` specs in open source software reveal about the utility of `clojure.spec`'s function semantics in practice?

I do not attempt to answer the following questions:

- How frequently do users instrument specs for runtime verification?

2.4 Methodology

2.4.1 Sources

To determine the frequency of `fdef` and `fspec` specs, the search features of `GitHub` [2] and `CrossClj` [3] were used.

GitHub indexes tens of thousands of open source Clojure projects, and provides a rudimentary search interface that is sufficient for discovering textual occurrences of functions and macros. False positives are common however, such as GitHub does not distinguish between "toy" projects and those with official releases, so we remove the former manually. This is because toy projects do not give a good indication of real-world idioms—for example, hundreds of projects simply contain experiments with `clojure.spec` that are not officially released or maintained.

CrossClj maintains a rich database of cross-links between Clojure projects. As of Febuary 2018, it indexes 9,438 projects, all of which have official releases (unlike GitHub search) and thus have more credibility that they are used. Cross-links are gathered for function/macro usages, and transitive project dependencies. Unlike GitHub, CrossClj also distinguishes between ClojureScript and Clojure code

2.4.2 Frequency Data gathering

Simple GitHub searches were used to find occurrences of `fdef` and `fspec`. As of March 2018, searches for `fdef` yield around 2,000 results [4], and for `fspec` [5] yield around 600 results. Searches were quite noisy, so less actual examples of these forms were found.

As a baseline, we searched for several common spec forms. The spec `def` form for defining spec aliases found around 4,400 results. The spec `keys` form for heterogeneous maps found around 3,500 results. All of these numbers are compared in Figure 2.

CrossClj function/macro cross-links were used to find occurrences of spec idioms. From the latest index (updated February 20th 2018) `fdef` occurs in 721 top-level forms over 83 projects [6], and `fspec` occurs in 22 top-level forms over 8 projects [7] (the mode number of occurrences per project was 1).

An error was found in the CrossClj data, however. To find a baseline number of projects using `clojure.spec`, we looked for occurrences of `s/def`, used

[2] `https://github.com`
[3] `https://crossclj.info`
[4] `https://github.com/search?q=fdef+language%3Aclojure&type=Code`
[5] `https://github.com/search?q=fspec+language%3Aclojure&type=Code`
[6] `https://crossclj.info/fun/clojure.spec.alpha/fdef.html`
[7] `https://crossclj.info/fun/clojure.spec.alpha/fspec.html`

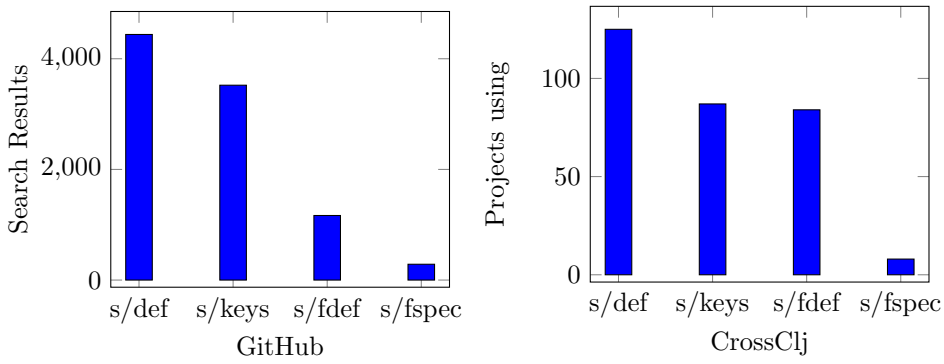


Figure 2: Left, the number of results for GitHub searches about spec forms. Right, the number of projects on CrossClj that use spec features. Searches for `ifn?` are omitted because it has been used as a common predicate since before 2009 (spec was released in 2017).

to define spec aliases. It should greatly outnumber the number of `fdef`'s in the ecosystem—we were surprised to find CrossClj reports the opposite. We identified that multiple occurrences of `s/def` in the same project are not always counted by CrossClj, so while CrossClj claims `s/def` occurs in more 60% more projects than `fdef` (125), CrossClj reports `s/def` only occurs in 349 top-level forms [8]. The latter number is too low—CrossClj reports, for example, the `clj-time` library only has 2 occurrences of `s/def`, but it actually has 7 occurrences.

We did not notice such an error in the counts for `fspec` and `fdef`, but this error reduces confidence on the exact numbers. Interestingly, the *relative* numbers of projects using these forms matches our intuition—`fspec` occurs 10 times less often than `fdef`, and `fdef` occurs about half as often as `s/def` (see plot in Figure 2). Still, these numbers should be treated with suspicion.

2.4.3 Project Samples

We selected 18 open source projects that use `fspec` to manually examine. They were chosen by first searching GitHub for `fspec` occurrences, and manually choosing the first few dozen results, manually keeping only projects seemed to have official releases. Figure 3 summarizes these findings. We have included the project names and our scratch notes as Appendix A.

We also searched for negative examples of `fspec` usage—that is, occurrences of higher-order functions with specs that did not use `fspec`—by searching for both `fdef` and `ifn?` [9], the flat contract for Clojure functions. We selected 17 projects using a similar method via a GitHub search. Appendix B contains the projects and our scratch notes for this experiment.

[8] `https://crossclj.info/fun/clojure.spec.alpha/def.html`

[9] `https://github.com/search?utf8=%E2%9C%93&q=ifn%3F+fdef+language%3Aclojure&type=Code`

	`fspec`	`ifn?`
Total occurrences in `fdef`	79	3
• Total occurrences in `fdef` arguments	65 (82%)	3
• Total occurrences in `fdef` return	14 (18%)	0
Total occurrences in map spec	41	2
• Total occurrences in heterogeneous map spec	40 (97%)	2
• Total occurrences in homogeneous map spec	1 (3%)	0

Figure 3: Function specs in practice, in 18 open source projects sourced from GitHub that utilized `fspec`.

2.5 Experiment 1: Frequency of `fspec`

Our first investigation concentrated on projects that use `fspec`. We chose 18 open source projects and manually investigated their usage of spec.

To gauge how `fspec`s are used in practice, we divided their occurrences into several categories. Figure 3 presents the results.

First, we measure how many times `fspec` occurs in top-level function specs (i.e., `fdef` forms). Because spec has different semantics for checking the argument and return of top-level function specs, we split this category into occurrences in the arguments spec and return spec. We also similarly measure the occurrences of `ifn?` in these projects, which was sparsely used in these projects.

We found 82% `fspec` instances occuring in the argument position, with 18% in the return position.

Second, we measure the frequency of `fspec` in a hash-map spec. Of the occurrences of `fspec` in map specifications, we found a strong preference for using heterogeneous map specs (97%), with the rest being homogeneous maps. If `fspec` occurs in a nested map, each map spec occurrence is counted. Similar to the previous experiment, the flat function contract `ifn?` was rarely used in map specs in these projects.

We noticed several interesting things while conducting these experiments.

There was only 2 occurrence of nesting function specs more than 2 deep—in a library that provides functional lenses [10]. Of those, one used 3 `fspec`s, the other 2 `fspec`s with a terminating `ifn?`.

We also noticed several users combining `fspec`s with the `or` spec for disjunctions. It's unclear how well this works in practice (and might only be there for documentation), but the "looser" generative testing semantics of `fspec` seems to have more compatibility with disjunction contracts than the "stricter" proxy-based verification approach.

Spec also provides the ability to write dependent function contracts via the `:fn` keyword option of `fspec`, allowing programmers to add custom code to verify the relationship between function arguments and return values. We found several interesting dependent contracts that used surprising techniques

[10] https://github.com/andrewmcveigh/bifocal

	`fspec`	`ifn?`
Total occurrences in `fdef`	0	188
• Total occurrences in `fdef` arguments	0	170 (90%)
• Total occurrences in `fdef` return	0	18 (10%)
Total occurrences in map spec	0	173
• Total occurrences in heterogeneous map spec	0	67 (39%)
• Total occurrences in homogeneous map spec	0	106 (61%)

Figure 4: Flat function specs in practice, in 17 open source projects sourced from GitHub that utilized `ifn?`. The 106 homogeneous map spec occurrences were sourced from only 3 of the projects, one project contributing the maximum 93 occurrences. Heterogeneous map specs occurrences were from 7 of the projects, with maximum 30 occurrences in one project.

like memoization to verify calls to higher-order functions [11].

Some developers explicitly worked around the `fspec`'s generative testing semantics. We found several comments and workarounds explaining why `fspec`s were not appropriate in particular contexts. Common concerns were

- triggering function side effects,
- difficultly writing generators for testing the function, and
- being unsure whether generative testing was appropriate.

We speculate the lack of polymorphic function specs contributed to these concerns. Since instances of a type variable were replaced with `any?` specs, writing a generator often did not make sense. It's unclear whether polymorphic specs are even feasible to check, but some developers at least seem attached to the documentation capabilities of `fspec`, perhaps suggesting more control over the testing semantics of `fspec` would be welcomed by spec users.

2.6 Experiment 2: Frequency of `ifn?`

In our second experiment, we measured the frequency of `ifn?` in 17 open source projects. The results are summarised in Figure 4.

Similar to our first experiment, occurrences of `ifn?` were mostly in the argument positions in `fdef` (90%). The data for `ifn?` occurrences in map specs is somewhat skewed, since one project contributed over half of the occurrences. However, even without taking that project into account, around half of all the `ifn?` occurrences were found in map specs.

One striking divide we noticed was the obvious preference for either `ifn?` or `fspec` per project. None of the projects that primarily used `ifn?` used `fspec` even once. In the first experiment, projects that used `fspec` used `ifn?` rarely—mostly to avoid `fspec`'s generative testing semantics.

[11] https://github.com/CharlesHD/chu.graph

2.7 Conclusions

Our goal in this section was to inform our decisions for when to generate `fspec` annotations in our automatic spec annotation tool. We surveyed open source projects via GitHub by measuring instances of two kinds of projects: those that primarily used `fspec`, and those that primarily used `ifn?`.

Our experiments yielded many interesting insights. Even using broad measurements like GitHub searches and CrossClj project references, we consistently found other common spec features were used roughly an order of magitude more than `fspec` (Figure 2).

However, we still investigated how programmers use `fspec` (Section 2.5). In the projects that primarily used `fspec` for first-class function specs, we found several instances of programmers enjoying the expressive documentation of `fspec`, but avoiding its generative testing semantics in creative ways. Sometimes `fspec`s commented out and replaced with `ifn?`, implying spec might benefit from more control over `fspec`'s testing semantics.

Our second experiment (Section 2.6) measured how programmers combined spec with `ifn?`. Interestingly, none of the 17 projects we selected used `fspec`—programmers seem divided in strong preferences for `ifn?` and `fspec`, with `fspec` users using `ifn?` only occasionally to work around `fspec`'s testing semantics.

A unique feature of spec's runtime instrumentation of `fdef`s is that return values are not checked. Since we predicted programmers might have issues with `fspec`'s generative testing semantics, we were interested if `fspec`'s were more frequent than usual in the return position of `fdef` (where these semantics would be implicitly suppressed). In our investigation, we found no strong evidence that programmers preferred `fspec` over `ifn?` in the return position—when they occurred in `fdef` forms, `fspec` was in the return position 18% of the time (Figure 3), and `ifn?` was 10% of the time (Figure 4).

One difference we found was `ifn?` occurs with a high correlation in a map spec (92% of the time), while `fspec` occurred in a map spec only 51% of the time. However, `fspec` occurred much more often in heterogeneous map specs (97% of occurrences in map specs were heterogeneous), while `ifn?` only occurred in heterogeneous maps 39% of the time. Heterogeneous maps in Clojure are used like records or structs in other languages [1], so the appeal of combining `fspec` with heterogeneous maps is similar to giving precise annotations to record or struct fields.

The lack of polymorphism in spec coupled with the frequency of polymorphic functions in Clojure seems to be a strong reason to prefer `ifn?` in many situations. We found `ifn?` was often used to avoid "any to any" function specs, which would otherwise use the generator for `any?` to generatively test the function value. This is inappropriate for many common variants on polymorphic functions like `map`, `filter`, and `reduce` and we found several examples in our experiments that support this view (see occurrences of "polymorphic" in Appendix B).

To tie back our investigation, what does this tell us about programmer preferences in an automatic spec annotation tool? At the very least, it tells us

there is no simple "fits all" behavior—we found preferences were in (at least) two distinct camps. This implies that the tool should have some customizability between preferring `fspec` and `ifn?`, but which should be the default? Should the tool prefer more expressive, but possibly incorrect specs, over less expressive but specs that "just work"?

I don't think there is a single definitive answer—rather each occurrence of `fspec` comes with its own context and tradeoffs. Whether it occurs in the arguments or return of an `fdef`, in a heterogeneous or homogeneous map, or even nested in another `fspec` are relevant contextual information that the default behavior of the tool should take into account. The varied and subtle results of this investigation support the conclusion that the tool's behavior should probably itself be varied and subtle.

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 (syntax defined in Figure 5). We extend λc with clojure.spec with `fdef` but without `fspec` specs, and call this model λc_s (syntax defined in Figure 6). Then, we extend λc_s to support `fspec` function contracts, and call this final model λc_s^f (syntax defined in Figure 7).

We define the small-step reduction rules ($\rightarrow$) for each language, using contexts. Figure 8 defines the reduction rules for the base language λc—it does not include any spec features. Figure 9 defines the reduction rules for λc_s, adding support for `fdef`. Finally, Figure 10 defines the reduction rules for λc_s^f, adding support for `fspec`.

3.2 Consistency property

Let $\rightarrow^*$ be the reflexive, transitive closure of $\rightarrow$, the single-step reduction relation for our respective languages.

We formulate two consistency properties, one which we expect to hold, another which does not hold, and test them both using Redex.

The first theorem (Theorem 1) states that any expression in λc evaluates to the same value (or error) as checking that value against any spec in λc_s, or throws a spec value. For example, `1` in λc evaluates to the same value (or error) as `(assert-spec 1 number?)`. Furthermore, `(assert-spec 1 zero?)` evaluates to a spec error.

Theorem 1 (Consistency without fspec)**.** *For every expression* E *in* λc*, and every spec* $\mathbb{S}$ *in* λc_s*, if* $E \rightarrow^* V_1^e$ *and* `(assert-spec` E $\mathbb{S}$`)` $\rightarrow^*$ V_2^e*, then either:*

- $V_1^e = V_2^e$*, or*
- V_2^e *is* `(error spec-error ...)`*.*

We formulated Theorem 1 in Redex, and tested it for 1000 expressions, each with 1000 specs. We found no counter-examples, as expected.

We expect to find a counter-example in Theorem 2, however. It is similar to Theorem 1, except we compare λc with λc_s, which includes `fspec`s (with generative testing semantics).

Theorem 2 (Consistency with fspec)**.** *For every expression* E *in* λc*, and every spec* $\mathbb{S}$ *in* λc_s^f*, if* $E \rightarrow^* V_1^e$ *and* `(assert-spec` E $\mathbb{S}$`)` $\rightarrow^*$ V_2^e*, then either:*

- $V_1^e = V_2^e$*, or*
- V_2^e *is* `(error spec-error ...)`*.*

Formulating Theorem 2 in Redex finds many counter-examples involving `fspec`. For example, the following call that generates only 10 expressions each with 10 random specs finds a "stuck" term from trying to generatively test a one-argument function `boolean?` as if it had zero-arguments (there is no way to know a function's arity in advance).

```
> (check-Clojure-ClojureSpecHOF-compat 10 10)
ERROR:
ClojureSpec evaluation did not fully reduce
Original-form: (assert-spec boolean? (FSpec () zero? 0))
Stuck-form: (assert-spec boolean? (FSpec () zero? 0))
```

3.3 Notes on Redex model

Caching was disabled for the Redex model because it interfered with generative testing. For example, the result of `(gen-spec number?)` was cached, so generative testing coverage was very poor.

The model rendered in this paper can be found at:

- `https://github.com/frenchy64/quals/blob/master/redex/clj2.rkt`.

For experimentation, the same model, but based on the Eval-Apply-Continue machine can be found at:

- `https://github.com/frenchy64/quals/blob/master/redex/clj.rkt`.

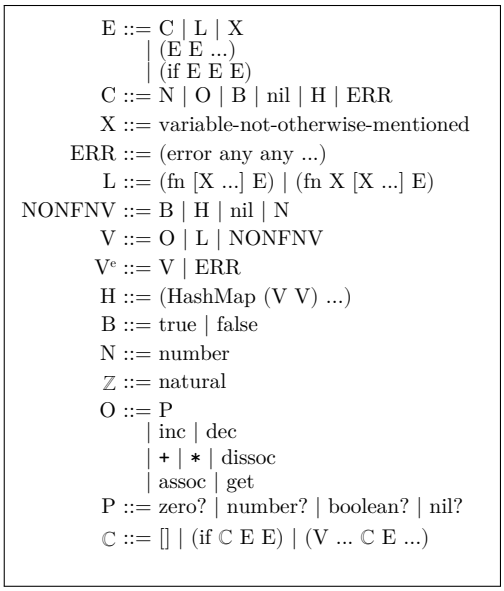


Figure 5: Syntax of Terms in λc. Expressions `E` consist of (loosely named) "constant" expressions `C` (numbers `N`, built-in functions `O`, booleans, nil, hash maps `H`, and errors `ERR`), functions `L` (non-recursive, and recursive), variables `X`, applications, and conditionals. The built-ins `assoc`, `dissoc`, and `get` perform the hash map operations add, remove, and lookup, respectively. Values are denoted `V`, and we use contexts $\mathbb{C}$ to define reduction rules.

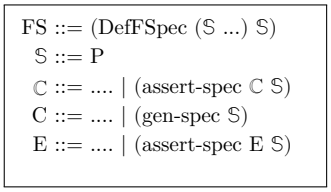


Figure 6: Syntax of λc_s (extending λc, Figure 5). We add the `assert-spec` form that takes an expression and a spec and checks the expression evaluates to a value conforming to the spec. We restrict specs to just predicates `P`.

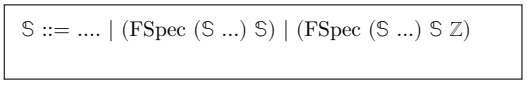


Figure 7: Syntax of λc_s^f (extending λc_s, Figure 6). We add two forms of `fspec`s—the natural number represents how many times to generatively test a function value.

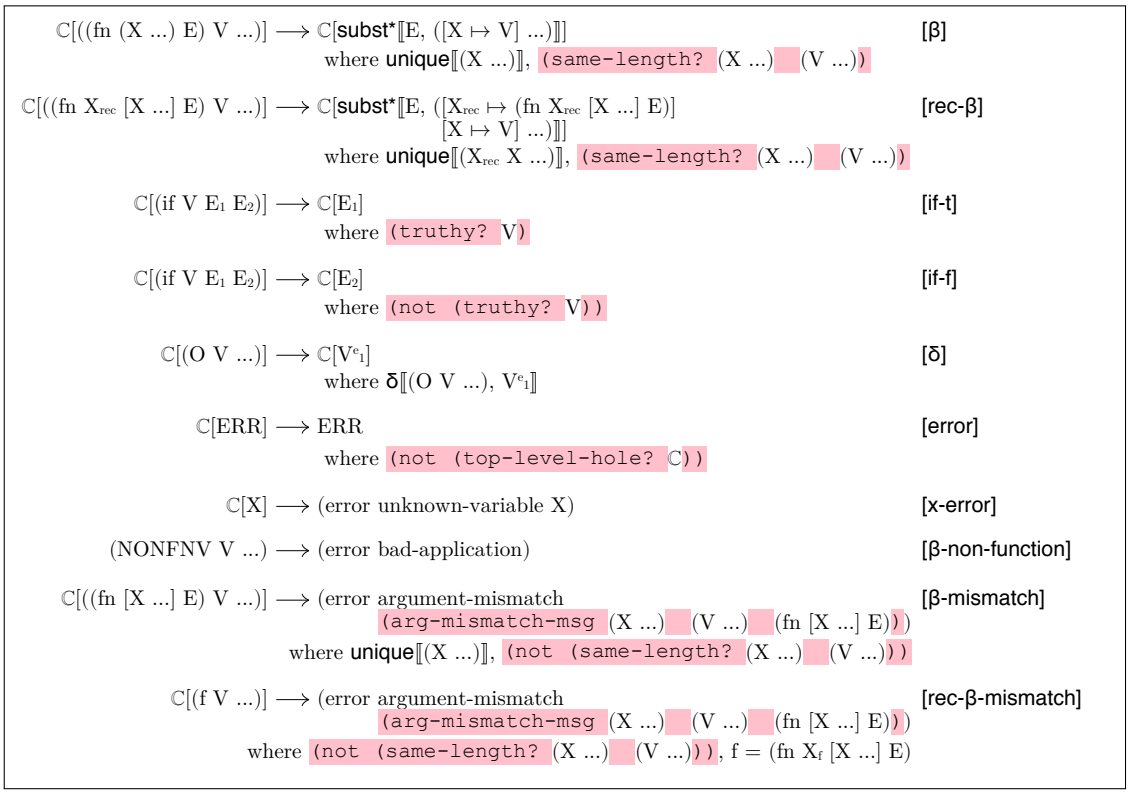


Figure 8: Small-step reduction relation in λc. We define β reduction rules for both types of functions. Then branching rules for conditionals (`false` and `nil` are false values), and constant functions (δ, full definition omitted). Finally several rules for throwing detailed runtime errors.

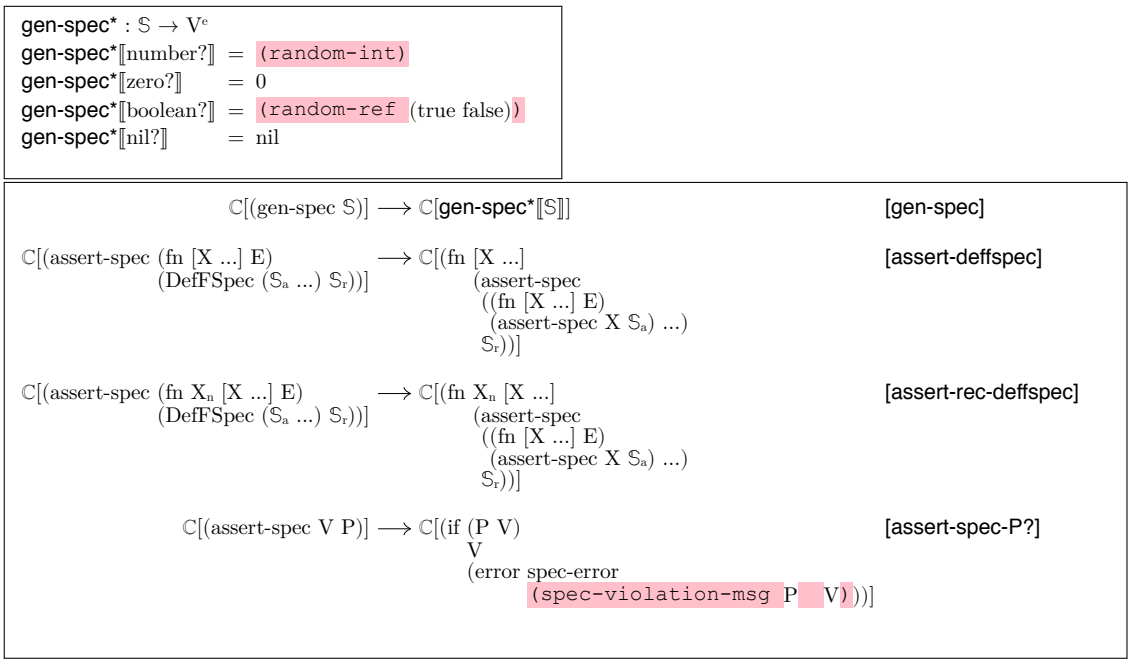


Figure 9: Small-step reduction relation in λc_s (extending λc, Figure 8). `gen-spec` takes a spec and generates a value conforming to that spec via the `gen-spec*` metafunction. We define `assert-spec` with support for for `fdef` using traditional proxy checking semantics, and flat predicates.

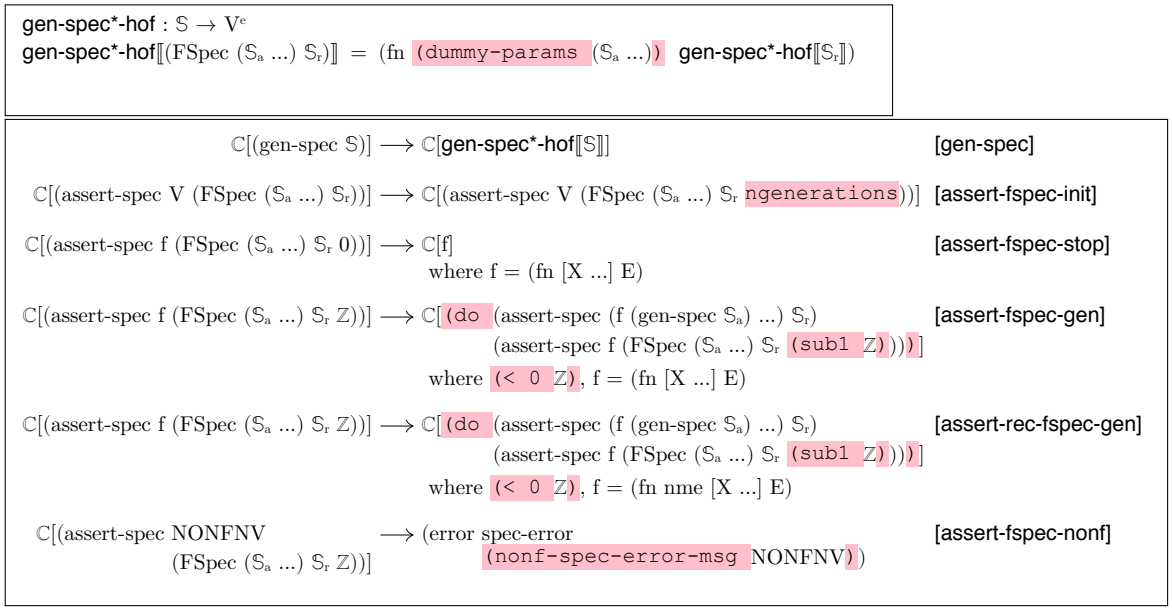


Figure 10: Small-step reduction relation for λc_s^f (extending λc_s, Figure 9) `gen-spec` takes a spec and generates a value conforming to that spec via the `gen-spec*-hof` metafunction (which extends `gen-spec*` in Figure 9). Since we add `fspec`s, `gen-spec*-hof` can generate values conforming to `fspec`s (dummy functions that have the expected number of parameters, and generate their return values). `assert-fspec-init` initializes `FSpec` with an initial number of generations. `assert-fspec-gen` and `assert-fspec-stop` handle the actual generative testing, with `assert-rec-fspec-stop` supporting recursive functions. Finally, `assert-fspec-nonf` handles non-function values that are expected to conform to an `fspec`.

References

[1] A. Bonnaire-Sergeant, R. Davies, and S. Tobin-Hochstadt. Practical optional types for clojure. In *European Symposium on Programming Languages and Systems*, pages 68–94. Springer, 2016.

[2] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical report, J. ACM, 2003.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

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

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

A Q2: Experiment 1

This section we put our notes on the first experiment.

```
valuehash:
- 1 fn returning fn
- 1 fn taking fn
- Note: fn takes input-stream, how to generate? are fspec generators used in ‘lein test‘?
https://github.com/arachne-framework/valuehash/blob/fdd19b4a4c3b294d46fe7e0b50187290043b48aa/src/valuehash/specs.clj

arachne-fileset: trying to get best of both worlds (documentation but no generative testing)
- 1 fn taking pred
- Note: fn takes File
https://github.com/arachne-framework/arachne-fileset/blob/0336d2d8d273eb1e0a862641000da1bd76099626/test/user.clj#L35

dspec: utility function, checks direct HOF
- 1 fn taking pred (filter wrapper)
https://github.com/lab-79/dspec/blob/26f88e74066e381c8569d175c1bd5948a8005bd0/src/clj/lab79/dspec/util.clj

async-connect: maps of fn handlers
- 19 fspecs, for a ‘keys‘ of functions
- Note: ‘with-generator‘ suppresses all fspec generators, but that doesn’t suppress
  instrumentation fn generative testing. Unsure if generative testing used.
  Takes a :netty/context, but that spec isn’t defined anywhere...
https://github.com/tyano/async-connect/blob/4f30801485b68e60fc5352b8a169b6f5829d2553/src/async_connect/netty/handler.clj
- 7 fspecs, maps of fn’s (channel ops)
- Note: ‘with-generator‘ suppresses generators
https://github.com/tyano/async-connect/blob/4f30801485b68e60fc5352b8a169b6f5829d2553/src/async_connect/server.clj#L46

devcards: direct HOF, also mixes ifn? checks
- 3 ifn?’s
  - 2 in map input
  - 1 fn taking fn
- 1 fspec
  - 1 fn returning fn
  - 1 fn taking map with a fn entry
- Note: alpha quality
https://github.com/olivergeorge/devcards-vs-clojure-spec/blob/385e332c21e57b097b56e899c09e99a260daf3ad/src/devcards_vs_clojure_spec/core_specs.cljc#L7

email-tool: map of fn’s
- 2 fspecs
  - map args to fn
  - 1 fspec is any -> any
  - 1 fspec has easy to generate things (strings etc.)
https://github.com/andrewzhurov/email-tool/blob/fd6bf979c5534315edf0d5e2ca762a879d0c9587/src/email_tool/parts.clj#L8

bifocal: lenses, everything is HOF, sorely missing polymorphism
- 6 fspecs
  - 4 are basically any -> any
  - fn taking fn taking fn (3 nestings)
   - (s/def ::upd-f (s/fspec :args (s/cat :s any? :f fn?) :ret any?))
- 1 fn?
  - as above, mixing with fspec
https://github.com/andrewmcveigh/bifocal/blob/850e79452f4f9bc6966768055acfc7aae6671f80/src/bifocal/lens.clj#L36

triboard: fn taking fn
- 1 fspec
  - posn -> any
https://github.com/QuentinDuval/triboard/blob/dc9d60197262857bbea0756a5a395ba248929961/src/cljs/triboard/view/frame.cljs#L15

yoose: fn taking handler fn
- 1 fspec
  - 1 fn taking fn (any -> any, event handler)
https://github.com/brianium/yoose/blob/6400ed9e20f8472411c6cb0185a392cda097a0b8/src/brianium/yoose/spec.clj#L9

cljs-fn: map of fn’s
- 2 fspecs (react)
  - map of fn’s
  - render db/id fn
  - thunk for end of row formatting
https://github.com/briansunter/cljs-hn/blob/a15bac4535fd88d6f79a80864a0301fe3d7d8d60/src/hackernews/components/list.cljs#L15

kxix.collect: fn taking handler/processing fn’s
- 2 fspecs
  - 2 fn taking fn’s (handlers)
- Note: alpha, no release but not a toy
https://github.com/MastodonC/kixi.collect/blob/8a5e6a0de041f5684602235be6466afa805be92d/src/kixi/collect/aggregate.clj#L15
```

```
frereth-common: or + fn's !!!
- 5 fspecs
 - 3 fn taking fn
 - 2 fn returning fn
 - odd combination of or + fspec's
 - most (any -> any) but comments are unsure if it can be more specific
  - perhaps polymorphism might help?
- Note: unreleased, discouraged from use, but not a toy
https://github.com/jimrthy/frereth-common/blob/ff59081b170984d25e8e8192d34348ce36f7296c/src/com/frereth/common/methods.cljc#L33-L36
- 5 fspecs
  - 3 fn taking fn (bytes -> any)
  - 2 fn returning fn
https://github.com/jimrthy/frereth-common/blob/88e57bb942334124f29be1b9405bbf04c9c2af08/src/com/frereth/common/aleph.clj#L98

datomic-spec: homogeneous map with fn vals
- 1 fspec (in testing code)
  - 1 fn returning homogeneous map of fn vals (thunks that return test.check generators)
https://github.com/lab-79/datomic-spec/blob/880ab123b49da8cc79c27cd78c9a2455b260e4b9/src/cljc/lab79/datomic_spec/gen_overrides.cljc#L6

mqtt: many fspecs, contain very specific args
- 10 fspecs
 - 1 fn returning fn (connection init fn)
 - 1 fn taking fn (handler)
 - 4 fn map of fn
 - 1 fspec fn intersection (I (nil Val -> Any) (Val nil -> Any))
 -
https://github.com/dvlopt/mqtt/blob/c7f2dcaf8d4df0a31460c16f24c5b402f21df655/src/dvlopt/mqtt/v3.clj

chu.graph: fn returning fn
- 11 fspecs
 - 14 function taking fn
 - 2 functions returns fn
 - extremely complicated dependent function specs
  - https://github.com/CharlesHD/chu.graph/blob/a820ef8456b44b1044d7f6cd9340a5504ad393de/src/chu/graph.cljc#L78-L84
  - Note: are these even used?
https://github.com/CharlesHD/chu.graph/blob/a820ef8456b44b1044d7f6cd9340a5504ad393de/src/chu/graph.cljc#L17

takelist: fn returning fn
- 1 fspec
  - 1 fn returning fn
https://github.com/alexanderkiel/takelist/blob/434ef6f6e05ca406c446b81fa5a77c7f0519c355/src/takelist/app.clj#L27

java.jdbc: seems dubious these are ever used, unless fn's are stubbed.
- 6 fspec
  - 1 fn taking fn
  - 7 fn taking map of fn
  - 1 fn returning fn
- 1 fspec commented out
  - improved custom generator needed (database ResultSet's)
https://github.com/clojure/java.jdbc/blob/64a79366fa464be75bdf4bdda133441b9d1efb26/src/main/clojure/clojure/java/jdbc/spec.clj#L124

sparkle: disjunction between map and fn
- 1 fspec (fdef return)
https://github.com/GradySimon/sparkle/blob/d5d82c37ab6be8359be7d3b5524d8b32dac452a1/src/sparkle/layer.clj#L9
 - commented out dependent :fn clause
https://github.com/GradySimon/sparkle/blob/d5d82c37ab6be8359be7d3b5524d8b32dac452a1/src/sparkle/spec.clj#L18-L24
```

B Q2: Experiment 2

This section we put our notes on the second experiment.

```
arachne-fileset : explicitly avoids fspec, comments out more expressive specs to avoid generative behaviour
- 5 fn taking ifn?
- 2 homogeneous map of ifn?
- Comment: ;; Need to override specs here so it doesn't try to gen when I instrument
https://github.com/arachne-framework/arachne-fileset/blob/0336d2d8d273eb1e0a862641000da1bd76099626/src/arachne/fileset/specs.clj#L7

z-com : uses ifn? (probably polymorphic 1 arg function), probably doesn't make sense to gen
- 4 fn taking ifn?
- 4 heterogeneous map of ifn?
https://github.com/aw7/z-com/blob/3180acf693f620bde5c7fb9d7c300e5deb02f88a/src/z_com/standard.cljs#L18

meiro : uses ifn? (unclear how to fspec, might be possible with very specific generators)
```

https://github.com/defndaines/meiro/blob/19f93996b87663fec5ed70c4966d114aa4855d6b/src/meiro/backtracker.clj#L17
- 2 fn taking ifn?
- 1 fn returning ifn?
https://github.com/defndaines/meiro/blob/f4fe98f8a54ffd0cc78a671de96bcd9727904c0c/src/meiro/core.clj#L201

comfy : utility library, uses ifn? for HOF's (19 occurrences in about a dozen very polymorphic function specs)
- 16 fn taking ifn?
- 3 fn returning ifn?
- possible transducer returns, like in (map f)
https://github.com/madstap/comfy/blob/bbca80f269a912a3a4914188d8dac29e5edaca0b/src/madstap/comfy.cljc

ferje : polymorphic "app(ly)" function uses ifn?
- 1 fn taking ifn?
 - polymorphic
https://github.com/chourave/ferje/blob/d8a3261309a994bdbaf6e3af29fc6d22c3e51844/src/ferje/util.clj#L33

huri : 12 occurrences of ifn? (here combines `s/or` and ifn?, so fspec probably not appropriate)
- 109 fn taking ifn?
- 30 heterogeneous map ifn?
- 93 homogeneous map ifn?
https://github.com/sbelak/huri/blob/fc98c5f1870f524c1e2662980085b6a258abd5cf/src/huri/core.clj#L159-L161

arche : ifn? takes a "User Defined Type" (keyword), perhaps hard to generate?
- 6 fn taking ifn?
- 6 heterogeneous map ifn?
https://github.com/troy-west/arche/blob/1c739d178cbc5e1e1f0ac67feb64da2f8e82e099/src/troy_west/arche/spec.clj#L36

conllu-clj : ifn? for `keyfn`, but can only be one of 2 predefined def's
- 1 fn taking ifn?
https://github.com/ysmiraak/conllu-clj/blob/6bc02c8f3a28dcea871c20fb965878b21fb0c5e5/src/conllu/eval.clj#L19
- 2 fn taking ifn?
- arbitrary transformation functions
https://github.com/ysmiraak/conllu-clj/blob/564e64a94cfde69f58dc37da183e735ebd5a07bb/src/conllu/parse.clj#L42

planck : TCP data handlers are ifn?
- 2 fn taking ifn?
https://github.com/mfikes/planck/blob/3bc8b174834cf413dbc7415f7af30955adcc27b0/planck-cljs/src/planck/socket/alpha.cljs#L11-L12

owlet : callback is ifn?
- 1 fn taking ifn?
- callback
https://github.com/codefordenver/owlet/blob/4864e0cbc7726501cc58a1362347f07f10524ed7/src/cljs/owlet/views/confirm.cljs#L12

sqlingvo : evaluation fn is ifn? + another (latter could be enum of fns tbh)
- 2 fn returning ifn?
- 2 heterogeneous map ifn?
- ifn's are interfaces to db
https://github.com/r0man/sqlingvo/blob/183014264e998366cdb906dbfe35a984c7d5443f/src/sqlingvo/db.cljc

proletariat : 2 HOF helpers with IFn (reduce, conj wrappers)
- 2 fn taking ifn?
- could be polymorphic
https://github.com/LiaisonTechnologies/proletariat/blob/2a9a8cb8185785cb1d12376da21ddb97d5e43d51/src/proletariat/core.clj#L566

mazes : ifn? for predicate arg (but rest of fn is also sparsely annotated with sequential?)
- 1 fn taking ifn?
https://github.com/amacdougall/mazes/blob/1766a5fb2a3bbfc3141f44c09a2477a1ec65edef/src/cljc/mazes/generators/wilson.cljc#L36

Arcadia : listener is ifn?
- 1 fn taking ifn?
https://github.com/arcadia-unity/Arcadia/blob/a0f1ee9f3d8a5b248bb415001d2d0cb2d27527db/Source/arcadia/internal/state.clj#L52

hive : callback is ifn?
- 1 fn taking ifn?
- 1 heterogeneous map ifn?
https://github.com/hiposfer/hive/blob/f4323cc6ddba894942ba37329d4a5f7f7f974024/src/hive/services/raw/location.cljs#L12

datacore : map of callbacks is ifn? vals, + 4 function inputs as ifn?
- 14 fn taking ifn?
- 11 fn returning ifn?
- 22 heterogeneous map ifn?
- 11 homogeneous map ifn?
https://github.com/stathissideris/datacore/blob/e4ab7f4822edfccca821fb8f4f9ec81a69e9d056/src/datacore/cells.clj#L59

ambiparse : predicate is ifn?
- 1 fn returning ifn?
- 2 heterogeneous map ifn?
https://github.com/brandonbloom/ambiparse/blob/eeb047878e4990a877810ac4805a45d8cfe9acfb/src/ambiparse/gll.clj#L175