Squash the work!

Inferring Recursive Type Annotations from Plain Data for Optional Type Systems

Ambrose Bonnaire-Sergeant

Sam Tobin-Hochstadt

Abstract

To type check a dynamically-typed program with an optional type system, type annotations must be added. This burden is sometimes large, and has put off real users attempting to migrate to existing optional type systems. When not discouraged, programmers often annotate tens of thousands of lines of code without assistance.

We present an approach to lighten the load on programmers moving to optional types. Our only requirement of existing programs is that they be runnable, with a suite of tests or examples. Given a running program, we instrument the execution, record type information, summarize it, and annotate the existing program with the recovered types.

We apply our approach to Clojure, a dynamically typed language with a culture of unit testing as well as both an existing optional type system and a contract system. Given a component under consideration, we instrument the source and analyze the behavior of the program while running unit tests. Equipped with this information, we summarize it by generating compact type specifications for all the functions in the component, including well-named type definitions. Our tool can also automatically generate contracts using the Clojure spec tool. Since Clojure relies heavily on ad-hoc data structures in the Lisp tradition, we describe an algorithm for automatically inferring recursive structural types from data examples, a challenge not considered in prior work.

Our approach, as must be the case for a testing-driven tool, is incomplete—programs may have too few unit tests, and untested execution paths can have differing type behavior. We therefore evaluate our tool by running it on real Clojure programs and then completing the porting to Typed Clojure. We find that while some changes are always needed, the generated types are valuable and the effort reduction is substantial.

1 Introduction

It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.—Alan Perlis

Optional type systems [6] extend existing untyped languages with type checking. For example, TypeScript [3] and Flow [1] extend JavaScript, Hack [2] extends PHP, and mypy [16] extends Python. They typically support existing syntax and idioms of their target languages.

Transitioning to an optional type system requires adding type annotations to existing code, a significant manual burden. This overhead has sparked interest in tooling to help create [4, 7, 10, 12, 20] and evolve [15] these annotations.

```
(defn nodes
  "Returns the number of nodes in a binary tree."
  [t] (case (:op t)
        :leaf 1
        :node (+ 1 (nodes (:left t))
                   (nodes (:right t)))))
(assert (= 3 (nodes
               {:op :node,
                :left {:op :leaf, :val 2},
                :right {:op :leaf, :val 3}})))
```

Figure 1. A typical use of maps in Clojure to represent records, that requires type annotations to check with Typed Clojure. Our tool will automatically annotate this program with a useful recursive type (Figure 3).

Clojure [14] is an untyped language that compiles to the Java Virtual Machine. Compared to languages already mentioned, it strongly encourages programming with plain data structures, and is a good example of implementing Perlis' advise in our opening quote. Clojure provides many functions and idioms around persistent, immutable hash-maps, including literal map syntax `{k v ...}`, interned *keywords* suitable both for map keys (e.g., `:a`, `:b`) and functions that look themselves up in a map (e.g., `(:a {:a 1}) => 1`), and a suite of functions to deeply transform, manipulate, and validate maps, with multimethods providing open extension. Typed Clojure [5] is an optional type system for Clojure designed to recognize these idioms given sufficient type annotations—which our tool assists the programmer in writing.

Maps in Clojure often replace records or objects, demonstrated in Figure 1: instead of representing a binary tree with `Node` and `Leaf` classes, they are encoded in maps with an explicit keyword *dispatch entry* (e.g., `:op`) to distinguish cases—`{:op :leaf, :val ...}` for instances of `Leaf`, and `{:op :node, :left ..., :right ...}` for `Node`.

This emphasis on maps has far-reaching implications for Typed Clojure. Types for maps (written `'{:op ':leaf, :val ...}` and `'{:op ':node, :left ..., :right ...}` for our examples) combine with ad-hoc union types and equirecursive type aliases in the type:

```
(defalias Tree
  (U '{:op ':node, :left Tree, :right Tree}
     '{:op ':leaf, :val Int}))
```

```
function nodes(t: {left: {op: string, val: number},
                   op: string,
                   right: {op: string, val: number}}
                | {op: string, val: number}) ...
```

Figure 2. TypeWiz's TypeScript annotations for Figure 1.

Porting Figure 1 to Typed Clojure involves writing a type definition `Tree`, and annotating `nodes` as

```
(ann nodes [Tree -> Int])
```

Existing annotation tools fail to generate these types since only classes are given recursive types, and so cannot infer recursive types for plain data, such as JSON or heterogeneous dictionaries. To demonstrate the current state-of-the-art in automatic annotation tools for optional type systems, we transliterate Figure 1 to JavaScript using plain objects. We use TypeWiz [21] to generate TypeScript annotations via dynamic analysis, as it is well maintained and generates comparable annotations to similar tools [4, 7, 10, 12, 15, 20].

Figure 2 shows the actual output of TypeWiz for a JavaScript translation of Figure 1. Unfortunately, the annotation is too specific: it only accepts trees of height 1 or 2. For fair comparison, even a class-based translation to JavaScript with a common `nodes` method yields a shallow annotation:

```
class Node {                 class Leaf {
  public left: Leaf;           public data: number;
  public right: Leaf;
  ...                          ...
}                            }
```

Since TypeWiz uses dynamic analysis, it is faithfully providing the exact types that are observed at runtime. Unfortunately, incrementally better test coverage does not quickly converge to useful annotations. For example, if we add an example of a nested `Node` on the just the left branch the annotation for the plain objects version is still not recursive (alarmingly, it instead grows linearly in the number of nodes), and the class-based version helpfully updates the type of `left` to `Leaf|Node`, but `right` still remains `Leaf`.

On the other hand, our approach recognizes Figure 1 as traversing recursively defined data with two cases, distinguished by the `:op` entry (Figure 3).

Our approach is sensitive enough to to compute optional keys for each constructor. Adding a unit test that includes a `:val` entry in a `:node` yields the annotation

```
(defalias Op
  (U (HMap :mandatory
           {:op ':node, :left Op, :right Op}
           :optional {:val Int})
     '{:op ':leaf, :val Int}))
```

```
(defalias Op
  (U '{:op ':node, :left Op, :right Op}
     '{:op ':leaf, :val Int}))
(ann nodes [Op -> Int])
```

Figure 3. Our tool's Typed Clojure annotation of Figure 1. **`defalias`** introduces an equirecursive type alias, **`U`** is a set-theoretic union type constructor, and `':node` is a singleton type containing just the keyword value `:node`.

```
(defalias Op
  (U (HMap :mandatory
           {:op ':node, :left Op, :right Op}
           :optional {:val Int})
     '{:op ':node3, :left Op, :mid Op, :right Op}
     '{:op ':leaf, :val Int}))
(ann nodes [Op -> Int])
(ann nodes' [Op -> Int])
```

Figure 4. Optional entries and combined information across functions. **`HMap`** is the expanded heterogeneous map type constructor, specifying both mandatory and optional entries.

Furthermore, we aggressively combine recursive data used in the same file. Given a 3-way node example with the key `:node3`, used as input to a distinct function `nodes'`, our tool generates the combined type shown in Figure 4.

Contributions

- We outline a generalized approach to automatically generating type annotations (Section 2).
- We describe a novel approach to reconstructing recursively defined structural records from fully unrolled examples in a formal model of our inference algorithm (Section 3).
- We show how to extend our approach with space-efficient and lazy runtime tracking (Section 4).
- We report our experience using this algorithm to generate types, tests, and contracts on several Clojure libraries and programs (Section 5).

2 Overview

Now that we have introduced the problem, we can flesh out our philosophy and overall approach. To concretize our discussion, Figure 5 demonstrates our tool's output when generating types for the 1,776 line file cljs.compiler.

Its main function is `emit`, which effectfully converts a map-based AST to JavaScript. The AST is created by functions in cljs.analyzer, a significantly larger 4,366 line Clojure file. Without inspecting cljs.analyzer, our tool annotates `emit` on line 23 with a recursive AST type `Op` (lines 1-12).

An important question to address is "how accurate are these annotations?". Unlike previous work in this area [4], we do not aim for soundness guarantees in our generated types. A significant contribution of our work is a tool that Clojure programmers can use to help learn about and specify their programs. In that spirit, we strive to generate annotations meeting more qualitative criteria. Each guideline by itself helps generate more useful annotations and, as we discuss in Section 5.1, they combine in interesting ways help to make up for shortcomings.

Choose recognizable names Assigning a good name for a type increases readability by succinctly conveying its purpose. Along those lines, a good name for the AST representation on lines 1-12 might be `AST` or `Expr`. However, these kinds of names can be very misleading when incorrect, so instead of guessing them, our tool takes a more consistent approach and generates *easily recognizable* names based on the type the name points to. Then, those with a passing familiarity with the data flowing through the program can quickly identify and rename them. For example,

- `Op` (lines 1-12) is chosen because `:op` is clearly the dispatch key (the `:op` entry is also helpfully placed as the first entry in each case to aid discoverability),
- `ColumnLineContextMap` (lines 13-15) enumerates the keys of the map type it points to,
- `NameShadowMap` and `FnScopeFnSelfNameNsMap` (referenced on lines 4 and 5) similarly, and
- `HMap49305` (lines 16-22) shows how our tool fails to give names to certain combinations of types (we discuss the severity of this particular situation in Section 5.1).

Favor compact annotations Literally translating runtime observations into annotations without compacting them leads to unmaintainable and impractical types resembling TypeWiz's annotation for `nodes` (Figure 2). To avoid this, we use optional keys where possible, like line 15, infer recursive types like `Op`, and reuse type aliases in function annotations, like `emit` and `emit-dot` (lines 23, 24). These processes of compacting annotations often makes them more general, which leads into our next goal.

Don't overspecify types Poor test coverage can easily skew the results of dynamic analysis tools, so we choose to err on the side of generalizing types where possible. Our opening example `nodes` (Figure 1) is a good example of this—our inferred type (Figure 3) is recursive, despite `nodes` only being tested with a tree of height 2. This has several benefits.

- We avoid exhausting the pool of easily recognizable names by generalizing types to communicate the general role of an argument or return position. For example, `emit-dot` (line 24) is annotated to take `Op`, but in reality accepts only a subset of `Op`. Programmers can combine the recognizability of `Op` with the suggestive name of `emit-dot` to decide whether, for instance, to split `Op` into smaller type aliases or add type casts in the definition of `emit-dot` to please the type checker (some libraries require more casts than others to type check, as discussed in Section 5.2).
- Generated Clojure spec annotations (an extension discussed in Section 4.3) are more likely to accept valid input with specs enabled, even with incomplete unit tests (we enable generated specs on several libraries in Section 5.3).
- Our approach becomes more amenable to extensions improving the running time of runtime observation without significantly deteriorating annotation quality, like lazy tracking (Section 4.2).

```
1 (defalias Op ; omitted some entries and 11 cases
2   (U (HMap :mandatory
3            {:op ':binding,
4             :info (U NameShadowMap
5                      FnScopeFnSelfNameNsMap), ...}
6            :optional
7            {:env ColumnLineContextMap, :init Op,
8             :shadow (U nil Op), ...})
9     '{:op ':const, :env HMap49305, ...}
10    '{:op ':do, :env HMap49305,
11      :ret Op, :statements (Vec Nothing), ...}
12    ...))
13 (defalias ColumnLineContextMap
14   (HMap :mandatory {:column Int, :line Int}
15         :optional {:context ':expr}))
16 (defalias HMap49305 ; omitted some extries
17   (U nil
18      '{:context ':statement, :column Int, ...}
19      '{:context ':return, :column Int, ...}
20      (t/HMap :mandatory
21              {:context ':expr, :column Int, ...}
22              :optional {...})))
23 (ann emit [Op -> nil])
24 (ann emit-dot [Op -> nil])
```

Figure 5. Sample raw output from our tool inferring types for cljs.compiler, a 1,776 line file defining the code generation phase of a production-quality compiler. Its AST format is inferred as `Op` (lines 1-12) with 22 recursive references (like lines 7, 8, 11) and 14 cases distinguished by `:op` (like lines 3, 9, 10), 5 of which have optional entries (like lines 6-8). To improve inference time, only the code emission unit tests were exercised (299 lines containing 39 assertions) which normally take 40 seconds to run, from which we generated 448 lines of types and 517 lines of specs in 2.5 minutes on a 2011 MacBook Pro (16GB RAM, 2.4GHz i5).

Our general approach to generating types is separated into two phases—**collection** and **inference**. The collection phase (Section 3.1), gathers observations about a running program. This is achieved by instrumenting the program and exercising it, usually by running its unit tests, with space-efficient tracking (Section 4.1) avoiding redundant traversals of values. The inference phase (Section 3.2) uses these runtime observations to generate the final type annotations, with recursive types, optional entries, and good names.

The first pass in the inference phase generates a naive type environment from runtime observations (described in Section 3.2.1). Types are fully unrolled, appearing similar to TypeWiz's final annotation for `nodes` (Figure 2)—except it is the starting point of our algorithm.

A key hypothesis in our algorithm is that functions in the same file operate on related data. Our inference is built to be used per-file and aggressively merges all apparently-related HMap types—firstly types immediately nested within each other (Section 3.2.2), and then across type aliases (Section 3.2.3). This often yields useful types in our benchmarks. This approach has limited success for libraries providing polymorphic functions, since unit tests may use sample data that are unrelated to the details of a function. Our algorithm excels with programs that mainly operate on one or two (possibly recursive) map-based data representations, which, outside of polymorphic libraries, are common in the Clojure ecosystem in our experience.

3 Formalism

We present λ_{track}, an untyped λ-calculus describing the essense of our approach to automatic annotations. We split our model into two phases: the collection phase collect that runs an instrumented program and collects observations, and an inference phase infer that derives type annotations from these observations that can be used to automatically annotate the program.

We define the top-level driver function annotate that connects both pieces. It says, given a program e and top-level variables $\overline{x}$ to infer annotations for, return an annotation environment Δ with possible entries for $\overline{x}$ based on observations from evaluating an instrumented e.

$$\begin{array}{l} \mathsf{annotate} : e, \overline{x} \rightarrow \Delta \\ \mathsf{annotate} = \mathsf{infer} \circ \mathsf{collect} \end{array}$$

To contextualize the presentation of these phases, we begin a running example: inferring the type of a top-level function f, that takes a map and returns its :a entry, based on the following usage.

```
define f = λm.(get m :a)
(f {:a 42}) => 42
```

Plugging this example into our driver function we get a candidate annotation for f:

$$\mathsf{annotate}((f\ \{\text{:a}\ 42\}), [f]) = \{f : [\{\text{:a}\ \mathsf{N}\} \rightarrow \mathsf{N}]\}$$

$$\begin{array}{lll}
v & ::= n \mid k \mid [\lambda x.e, \rho]_{\mathsf{c}} \mid \{\overline{k\ v}\} \mid c & \text{Values} \\
e & ::= x \mid v \mid (\textbf{track}\ e\ \pi) \mid \lambda x.e & \\
 & \quad \mid \{\overrightarrow{e\ e}\} \mid (e\ \overline{e}) & \text{Expressions} \\
\rho & ::= \{\overline{x \mapsto v}\} & \text{Runtime environments} \\
l & ::= x \mid \textbf{dom} \mid \textbf{rng} \mid \textbf{key}_{m}(k) & \text{Path Elements} \\
\pi & ::= \overline{l} & \text{Paths} \\
r & ::= \{\overline{\tau_{\pi}}\} & \text{Inference results} \\
\tau, \sigma & ::= \mathsf{N} \mid [\tau \rightarrow \tau] \mid (\mathsf{HMap}^{m}_{o}) \mid (\bigcup\ \overline{\tau}) & \\
 & \quad \mid a \mid k \mid \mathbf{K} \mid \top \mid (\textbf{Map}\ \tau\ \tau) \mid ? & \text{Types} \\
\Gamma & ::= \{\overline{x : \tau}\} & \text{Type environments} \\
m, o & ::= \{\overline{k\ \tau}\} & \text{HMap entries} \\
A & ::= \{\overline{a \mapsto \tau}\} & \text{Type alias environments} \\
\Delta & ::= (A, \Gamma) & \text{Annotation environments}
\end{array}$$

Figure 6. Syntax of Terms, Types, Inference results, and Environments for λ_{track}

3.1 Collection phase

Now that we have a high-level picture of how these phases interact, we describe the syntax and semantics of λ_{track}, before presenting the details of collect. Figure 6 presents the syntax of λ_{track}. Values v consist of numbers n, Clojure-style keywords k, closures $[\lambda x.e, \rho]_{\mathsf{c}}$, constants c, and keyword keyed hash maps $\{\overrightarrow{k\ v}\}$.

Expressions e consist of variables x, values, functions, maps, and function applications. The special form $(\textbf{track}\ e\ \pi)$ observes e as related to path π. Paths π record the source of a runtime value with respect to a sequence of path elements l, always starting with a variable x, and are read left-to-right. Other path elements are a function domain **dom**, a function range **rng**, and a map entry $\textbf{key}_{\overline{k_1}}(k_2)$ which represents the result of looking up k_2 in a map with keyset $\overline{k_1}$.

Inference results $\{\overline{\tau_{\pi}}\}$ are pairs of paths π and types τ that say the path π was observed to be type τ. Types τ are numbers N, function types $[\tau \rightarrow \tau]$, ad-hoc union types $(\bigcup\ \tau\ \tau)$, type aliases a, and unknown type $?$ that represents a temporary lack of knowledge during the inference process. Heterogeneous keyword map types $\{\overline{k\ \tau}\}$ for now represent a series of required keyword entries—we will extend them to have optional entries in later phases.

The big-step operational semantics $\rho \vdash e \Downarrow v\ ;\ r$ (Figure 7) says under runtime environment ρ expression e evaluates to value v with inference results r. Most rules are standard, with extensions to correctly propagate inference results r. B-Track is the only interesting rule, which instruments its fully-evaluated argument with the track metafunction.

The metafunction $\mathsf{track}(v, \pi) = v'\ ; r$ (Figure 7) says if value v occurs at path π, then return a possibly-instrumented v' paired with inference results r that can be immediately derived from the knowledge that v occurs at path π. It has a case for every kind of value. The first three cases records the number input as type N. The fourth case, for closures, returns a wrapped value resembling higher-order function contracts [9], but we track the domain and range rather than

B-Track

$$\frac{\rho \vdash e \Downarrow v \,;\, r \qquad \text{track}(v, \pi) = v' \,;\, r'}{\rho \vdash (\textbf{track}\ e\ \pi) \Downarrow v' \,;\, r \cup r'}$$

B-App

$$\frac{\rho \vdash e_1 \Downarrow [\lambda x.e, \rho']_{\text{c}} \,;\, r_1 \qquad \rho \vdash e_2 \Downarrow v \,;\, r_2 \qquad \rho'[x \mapsto v] \vdash e \Downarrow v' \,;\, r_3}{\rho \vdash (e_1\ e_2) \Downarrow v' \,;\, \bigcup \overline{r_i}}$$

B-Clos

$$\rho \vdash \lambda x.e \Downarrow [\lambda x.e, \rho]_{\text{c}} \,;\, \{\}$$

B-Delta

$$\frac{\rho \vdash e \Downarrow c \,;\, r_1 \qquad \overrightarrow{\rho \vdash e' \Downarrow v \,;\, r'} \qquad \delta(c, \overline{v}) = v' \,;\, r_2}{\rho \vdash (e\ \overrightarrow{e'}) \Downarrow v' \,;\, \overrightarrow{r \cup r'}}$$

B-Val

$$\rho \vdash v \Downarrow v \,;\, \{\}$$

B-Var

$$\rho \vdash x \Downarrow \rho(x) \,;\, \{\}$$

$$\begin{aligned}
\text{track}(n, \pi) &= n \,;\, \{\mathsf{N}_\pi\} \\
\text{track}(k, \pi) &= k \,;\, \{\mathbf{K}_\pi\} \\
\text{track}(c, \pi) &= c \,;\, \{\} \\
\text{track}([\lambda x.e, \rho]_{\text{c}}, \pi) &= [e', \rho]_{\text{c}} \,;\, \{\} \\
&\quad \text{where } y \text{ is fresh,} \\
&\quad e' = \lambda y.(\textbf{track}\ ((\lambda x.e)\ (\textbf{track}\ y\ \pi :: [\textbf{dom}])) \\
&\qquad\qquad\qquad \pi :: [\textbf{rng}]) \\
\text{track}(\{\}, \pi) &= \{\} \,;\, \{\{\}_\pi\} \\
\text{track}(\{\overline{k_1\ k_2}\ \overline{k\ v}\}, \pi) &= \{\overline{k_1\ k_2}\ \overline{k\ v'}\} \,;\, \bigcup r \\
&\quad \text{where } \overline{\text{track}(v, \pi :: [\textbf{key}_{\{\overline{k_1\ k_2}\ \overline{k\ ?}\}}(k)]) = v' \,;\, r}
\end{aligned}$$

$$\begin{aligned}
\delta(\text{assoc}, \{\overline{k\ v}\}, k', v') &= \{\overline{k\ v}\}[k' \mapsto v'] \,;\, \{\} \\
\delta(\text{get}, \{k\ v, \overline{k'\ v'}\}, k) &= v \,;\, \{\} \\
\delta(\text{dissoc}, \{k\ v, \overline{k'\ v'}\}, k) &= \{\overline{k'\ v'}\} \,;\, \{\}
\end{aligned}$$

Figure 7. Operational semantics, $\text{track}(v, \pi) = v \,; r$ and constants

verify them. The remaining rules case, for maps, recursively tracks each map value, and returns a map with possibly wrapped values. Immediately accessible inference results are combined and returned. A specific rule for the empty map is needed because we otherwise only rely on recursive calls to **track** to gather inference results—in the empty case, we have no data to recur on.

Now we have sufficient pieces to describe the initial collection phase of our model. Given an expression e and variables $\overline{x}$ to track, $\text{instrument}(e, \overline{x}) = e'$ returns an instrumented expression e' that tracked usages of $\overline{x}$. It is defined via capture-avoiding substitution:

$$\text{instrument}(e, \overline{x}) = e[\overline{(\textbf{track}\ x\ [x])/x}]$$

Then, the overall collection phase $\text{collect}(e, \overline{x}) = r$ says, given an expression e and variables $\overline{x}$ to track, returns inference results r that are the results of evaluating e with instrumented occurrences of $\overline{x}$. It is defined as:

$$\text{collect}(e, \overline{x}) = r, \text{ where } \vdash \text{instrument}(e, \overline{x}) \Downarrow v \,;\, r$$

For our running example of collecting for the program $(f\ \{:\!a\ 42\})$, we instrument the program by wrapping occurrences of f with **track** with path $[f]$.

$$\text{instrument}((f\ \{:\!a\ 42\}), [f]) = ((\textbf{track}\ f\ [f])\ \{:\!a\ 42\})$$

Then we evaluate the instrumented program and derive two inference results (colored in red for readability):

$$\vdash ((\textbf{track}\ f\ [f])\ \{:\!a\ 42\}) \Downarrow 42 \,;\, \{\mathsf{N}_{[f, \textbf{dom}, \textbf{key}(:a)]}, \mathsf{N}_{[f, \textbf{rng}]}\}$$

Here is the full derivation:

$$\begin{aligned}
&\texttt{=>}\ ((\texttt{track}\ f\ [f])\ \{:\!a\ 42\}) \\
&\texttt{=>}\ (\texttt{track}\ (\text{get}\ (\texttt{track}\ \{:\!a\ 42\}\ [f,\ \texttt{dom}])\ :\!a)\ [f,\ \texttt{rng}]) \\
&\texttt{=>}\ (\texttt{track}\ (\text{get}\ \{:\!a\ 42\}\ ;\ \{\mathsf{N}_{[f,\ \texttt{dom},\ \texttt{key}(:a)]}\}\ :\!a)\ [f,\ \texttt{rng}]) \\
&\texttt{=>}\ (\texttt{track}\ 42\ ;\ \{\mathsf{N}_{[f,\ \texttt{dom},\ \texttt{key}(:a)]}\}\ [f,\ \texttt{rng}]) \\
&\texttt{=>}\ 42\ ;\ \{\mathsf{N}_{[f,\ \texttt{dom},\ \texttt{key}(:a)]},\ \mathsf{N}_{[f,\ \texttt{rng}]}\}
\end{aligned}$$

Notice that intermediate values can have inference results (colored) attached to them with a semicolon, and the final value has inference results about both f's domain and range.

3.2 Inference phase

After the collection phase, we have a collection of inference results r which can be passed to the metafunction $\text{infer}(r) = \Delta$ to produce an annotation environment:

$$\begin{aligned}
&\text{infer} : r \rightarrow \Delta \\
&\text{infer} = \text{inferRec} \circ \text{toEnv}
\end{aligned}$$

The first pass $\text{toEnv}(r) = \Gamma$ generates an initial type environment from inference results r. The second pass

$$\text{squashLocal}(\Gamma) = \Delta'$$

creates individual type aliases for each HMap type in Γ and then merges aliases that both occur inside the same nested type into possibly recursive types. The third pass $\text{squashGlobal}(\Delta) = \Delta'$ merges type aliases in Δ based on their similarity.

3.2.1 Pass 1: Generating initial type environment

The first pass is given in Figure 8. The entry point toEnv folds over inference results to create an initial type environment via update. This style is inspired by occurrence typing [22], from which we also borrow the concepts of paths into types.

We process paths right-to-left in update, building up types from leaves to root, before joining the fully constructed type with the existing type environment via $\sqcup$. The first case handles the **key** path element. The extra map of type information preserves both keyset information and any entries that might represent tags (populated by the final case of **track**, Figure 7). This information helps us avoid prematurely collapsing tagged maps, by the side condition of the HMap $\sqcup$ case. The $\sqcup^{\text{H}}$ metafunction aggressively combines two HMaps—required keys in both maps are joined and stay required, otherwise keys become optional.

The second and third update cases update the domain and range of a function type, respectively. The $\sqcup$ case for function types joins covariantly on the domain to yield more useful annotations. For example, if a function accepts N and $\mathbf{K}$, it will have type $[\mathsf{N} \rightarrow ?] \sqcup [\mathbf{K} \rightarrow ?] = [(\bigcup\ \mathsf{N}\ \mathbf{K}) \rightarrow ?]$.

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\sqcup : \tau, \tau \to \tau} \\
(\bigcup\ \overline{\sigma}) \sqcup \tau & = & (\bigcup\ \overline{\sigma \sqcup \tau}) \\
\tau \sqcup (\bigcup\ \overline{\sigma}) & = & (\bigcup\ \overline{\sigma \sqcup \tau}) \\
? \sqcup \tau & = & \tau \\
\tau \sqcup ? & = & \tau \\
[\tau_1 \to \sigma_1] \sqcup [\tau_2 \to \sigma_2] & = & [\tau_1 \sqcup \tau_2 \to \sigma_1 \sqcup \sigma_2] \\
(\mathsf{HMap}^{m_1}_{o_1}) \sqcup (\mathsf{HMap}^{m_2}_{o_2}) & = & (\mathsf{HMap}^{m_1}_{o_1}) \sqcup^{\mathsf{H}} (\mathsf{HMap}^{m_2}_{o_2}), \\
 & & \overline{(k, k_i) \in m_i} \Rightarrow \overline{k_{i-1} = k_i} \\
\tau \sqcup \sigma & = & (\bigcup\ \tau\ \sigma), \text{otherwise}
\end{array}
$$

$$
\begin{array}{l}
(\mathsf{HMap}^{m_1}_{o_1}) \sqcup^{\mathsf{H}} (\mathsf{HMap}^{m_2}_{o_2}) = (\mathsf{HMap}^{m}_{o}) \\
\quad \text{where} \quad \mathsf{req} = \bigcup \overline{\mathsf{dom}(m_i)} \\
\qquad\qquad \mathsf{opt} = \bigcup \overline{\mathsf{dom}(o_i)} \\
\qquad\qquad \overline{k^r} = \bigcap \overline{\mathsf{dom}(m_i)} \setminus \mathsf{opt} \\
\qquad\qquad \overline{k^o} = \mathsf{opt} \cup (\mathsf{req} \setminus \overline{k^r}) \\
\qquad\qquad m = \overline{\{k^r\ \bigsqcup \overline{m_i[k^r]}\}} \\
\qquad\qquad o = \overline{\{k^o\ \bigsqcup \overline{m_i[k^o], o_i[k^o]}\}}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{fold} : \forall \alpha, \beta.(\alpha, \beta \to \alpha), \alpha, \overline{\beta} \to \alpha \\
\mathsf{fold}(\mathsf{f}, \mathsf{a}_0, \overline{\mathsf{b}}^{n}) = \mathsf{a}_n \\
\quad \text{where } \overline{\mathsf{a}_i = \mathsf{f}(\mathsf{a}_{i-1}, \mathsf{b}_i)}^{1 \le i \le n} \\
\\
\mathsf{toEnv} : r \to \Gamma \\
\mathsf{toEnv}(r) = \mathsf{fold}(\mathsf{update}, \{\}, r)
\end{array}
\qquad
\begin{array}{lcl}
\multicolumn{3}{l}{\mathsf{update} : \Gamma, \tau_\pi \to \Gamma} \\
\mathsf{update}(\Gamma, \tau_{\pi::[\mathbf{key}_{\{\overline{k'\ \sigma}\}}(k)]}) & = & \mathsf{update}(\Gamma, \{\overline{k'\ \sigma}\ k\ \tau\}_\pi) \\
\mathsf{update}(\Gamma, \tau_{\pi::[\mathbf{dom}]}) & = & \mathsf{update}(\Gamma, [\tau \to ?]_\pi) \\
\mathsf{update}(\Gamma, \tau_{\pi::[\mathbf{rng}]}) & = & \mathsf{update}(\Gamma, [? \to \tau]_\pi) \\
\mathsf{update}(\Gamma[x \mapsto \sigma], \tau_{[x]}) & = & \Gamma[x \mapsto \tau \sqcup \sigma] \\
\mathsf{update}(\Gamma, \tau_{[x]}) & = & \Gamma[x \mapsto \tau]
\end{array}
$$

Figure 8. Definition of $\mathsf{toEnv}(r) = \Gamma$

Returning to our running example, we now want to convert our inference results

$$r = \{\mathsf{N}_{[f, \mathbf{dom}, \mathbf{key}(:\mathrm{a})]}, \mathsf{N}_{[f, \mathbf{rng}]}\}.$$

into a type environment. Via $\mathsf{toEnv}(r)$, we start to trace $\mathsf{update}(\{\}, \mathsf{N}_{[f, \mathbf{dom}, \mathbf{key}(:\mathrm{a})]})$

3.2.2 Pass 2: Squash locally

We now describe the algorithm for generating recursive type aliases. The first step squashLocal creates recursive types from directly nested types. It folds over each type in the type environment, first creating aliases with aliasHMap, and then attempting to merge these aliases by squashAll.

A type is aliased by aliasHMap either if it is a union containing a HMap, or a HMap that is not a member of a union. While we will use the structure of HMaps to determine when to create a recursive type, keeping surrounding type information close to HMaps helps create more compact and readable recursive types. The implementation uses a post-order traversal via postwalk, which also threads an annotation environment as it applies the provided function.

Then, squashAll follows each alias a_i reachable from the type environment and attempts to merge it with any alias reachable from a_i. The squash function maintains a set of already visited aliases to avoid infinite loops.

The logic for merging aliases is contained in mergeAliases. Merging a_2 into a_1 involves mapping a_2 to a_1 and a_1 to the join of both definitions. Crucially, before joining, we rename occurrences of a_2 to a_1. This avoids a linear increase in the width of union types, proportional to the number of merged aliases. The running time of our algorithm is proportional to the width of union types (due to the quadratic combination of unions in the join function) and this optimization greatly helped the running time of several benchmarks. To avoid introducing infinite types, top-level references to other aliases we are merging with are erased with the helper f.

The merge? function determines whether two types are related enough to warrant being merged. We present our current implementation, which is simplistic, but is fast and effective in practice, but many variations are possible. Aliases are merged if they are all HMaps (not contained in unions), that contain a keyword key in common, with possibly disjoint mapped values. For example, our opening example has the `:op` key mapped to either `:leaf` or `:node`, and so aliases for each map would be merged. Notice again, however, the join operator does not collapse differently-tagged maps, so they will occur recursively in the resulting alias, but separated by union.

Even though this implementation of merge? does not directly utilize the aliased union types carefully created by aliasHMap, they still affect the final types. For example, squashing `T` in

```
(defalias T
  (U nil '{:op :node :left '{:op :leaf ...} ...}))
```

results in

```
(defalias T
  (U nil '{:op :node :left T ...} '{:op :leaf ...}))
```

rather than

```
(defalias T2 (U '{:op :node :left T ...}
                '{:op :leaf ...}))
(defalias T (U nil T2))
```

An alternative implementation of merge? we experimented with included computing sets of keysets for each alias, and

$$
\begin{aligned}
&\mathsf{aliasHMap} : \Delta, \tau \to (\Delta, \tau)\\
&\mathsf{aliasHMap}(\Delta, \tau) = \mathsf{postwalk}(\Delta, \tau, \mathsf{f})\\
&\quad \text{where} \quad \mathsf{f}(\Delta, (\mathsf{HMap}^{m_1}_{m_2})) = \mathsf{reg}(\Delta, (\mathsf{HMap}^{m_1}_{m_2}))\\
&\qquad\qquad\quad \mathsf{f}(\Delta, (\textstyle\bigcup\ \overline{\tau})) = \mathsf{reg}(\Delta, (\textstyle\bigcup\ \overline{\mathsf{resolve}(\tau)})),\\
&\qquad\qquad\quad\ \text{if } a \in \overline{\tau}\\
&\qquad\qquad\quad \mathsf{f}(\Delta, \tau) = (\Delta, \tau), \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{aliases} : \tau \to \overline{a}\\
&\mathsf{aliases}(a) = [a]\\
&\mathsf{aliases}(\tau(\overline{\sigma})) = \textstyle\bigcup \overline{\mathsf{aliases}(\sigma)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{postwalk} : \Delta, \tau, (\Delta, \tau \to (\Delta, \tau)) \to (\Delta, \tau)\\
&\mathsf{postwalk}(\Delta_0, \tau(\overline{\sigma}^n), \mathsf{w}) = \mathsf{w}(\Delta_n, \tau(\overline{\sigma'}))\\
&\quad \text{where} \quad \overline{(\Delta_i, \sigma'_i) = \mathsf{postwalk}(\Delta_{i-1}, \sigma_i, \mathsf{w})}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{mergeAliases} : \Delta, \overline{a} \to \Delta\\
&\mathsf{mergeAliases}(\Delta, []) = \Delta\\
&\mathsf{mergeAliases}(\Delta, [a_1 ... a_n]) = \Delta[\overline{a_i \mapsto a_1}][a_1 \mapsto \sigma]\\
&\quad \text{where} \quad \sigma = \textstyle\bigsqcup \overline{\mathsf{f}(\mathsf{resolve}(\Delta, a_i))[a_1/a_i]}\\
&\qquad\qquad\ \mathsf{f}(a') = (\textstyle\bigcup), \text{ if } a' \in \overline{a}\\
&\qquad\qquad\ \mathsf{f}((\textstyle\bigcup\ \overline{\tau})) = (\textstyle\bigcup\ \overline{\mathsf{f}(\tau)})\\
&\qquad\qquad\ \mathsf{f}(\tau) = \tau, \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{squashLocal} : \Gamma \to \Delta\\
&\mathsf{squashLocal}(\Gamma) = \mathbf{fold}(\mathsf{h}, (\{\}, \{\}), \Gamma)\\
&\quad \text{where} \quad \mathsf{h}(\Delta, x : \tau) = \Delta_2[x \mapsto \tau_2]\\
&\qquad\qquad\quad \text{where} \quad (\Delta_1, \tau_1) = \mathsf{aliasHMap}(\Delta, \tau)\\
&\qquad\qquad\qquad\qquad\ (\Delta_2, \tau_2) = \mathsf{squashAll}(\Delta_1, \tau_1)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{reg} : \Delta, \tau \to (\Delta, \tau)\\
&\mathsf{reg}(\Delta, \tau) = (\Delta[a \mapsto \tau], a), \ \text{where } a \text{ is fresh}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{resolve} : \Delta, \tau \to \tau\\
&\mathsf{resolve}(\Delta, a) = \mathsf{resolve}(\Delta[a])\\
&\mathsf{resolve}(\Delta, \tau) = \tau, \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{squashAll} : \Delta, \tau \to \Delta\\
&\mathsf{squashAll}(\Delta_0, \tau) = \Delta_n\\
&\quad \text{where} \quad \overline{a}^n = \mathsf{aliases}(\tau)\\
&\qquad\qquad\ \overline{\Delta_i = \mathsf{squash}(\Delta_{i-1}, [a_i], [])}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{squash} : \Delta, \overline{a}, \overline{a} \to \Delta\\
&\mathsf{squash}(\Delta, [], \mathsf{d}) = \Delta\\
&\mathsf{squash}(\Delta, a_1 :: \mathsf{w}, \mathsf{d}) = \mathsf{squash}(\Delta', \mathsf{w} \cup \mathsf{as}, \mathsf{d} \cup \{a_1\})\\
&\quad \text{where} \quad \mathsf{as} = \mathsf{aliases}(\Delta[a_1]) \setminus \mathsf{d}\\
&\qquad\qquad\ \mathsf{ap} = \mathsf{d} \setminus \{a_1\}\\
&\qquad\qquad\ \mathsf{f}(\Delta, a_2) = \ \text{if } \neg\mathsf{merge?}(\overline{\mathsf{resolve}(\Delta, a)}),\\
&\qquad\qquad\qquad\qquad\quad \text{then } \Delta\\
&\qquad\qquad\qquad\qquad\quad \text{else } \mathsf{mergeAliases}(\Delta, \overline{a_i})\\
&\qquad\qquad\ \Delta' = \ \text{if } a \in \mathsf{d}, \text{then } \Delta,\\
&\qquad\qquad\qquad\quad \text{else } \mathsf{fold}(\mathsf{f}, \Delta, \mathsf{ap} \cup \mathsf{as})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{merge?} : \overline{\tau} \to \mathbf{Bool}\\
&\mathsf{merge?}(\overline{(\mathsf{HMap}^{m_i}_{o_i})}) = \exists k.\overline{(k, k_i) \in m_i}\\
&\mathsf{merge?}(\overline{\tau}) = \mathbf{F}, \ \text{otherwise}
\end{aligned}
$$

Figure 9. Definition of $\mathsf{squashLocal}(\Gamma) = \Delta$

merging if the keysets overlapped. This, and many of our early experimentations, required expensive computations of keyset combinations and traversals over them that could be emulated with cruder heuristics like the current implementation.

3.2.3 Pass 3: Squash globally

The final step combines aliases without restriction on whether they occur "together". This step combines type information between different positions (such as in different arguments or functions) so that any deficiencies in unit testing coverage are massaged away.

The squashGlobal function is the entry point in this pass, and is similar in structure to the previous pass. It first creates aliases for each HMap via aliasSingleHMap. Then, HMap aliases are grouped and merged in squashHorizontally.

The aliasSingleHMap function first traverses the type environment to create HMap aliases via singleHMap, and binds the resulting envionment as Δ'. Then, alias environment entries are updated with f, whose first case prevents re-aliasing a top-level HMap, before we call singleHMap (singleHMap's second argument accepts both x and a). The $\tau(\overline{\sigma})$ syntax represents a type τ whose constructor takes types $\overline{\sigma}$.

After that, squashHorizontally creates groups of related aliases with groupSimilarReq. Each group contains HMap aliases whose required keysets are similar, but are never differently-tagged. The code creates a map r from keysets to groups of HMap aliases with that (required) keyset. Then, for every keyset $\overline{k}$, similarReq adds aliases to the group whose keysets are a subset of $\overline{k}$. The number of missing keys permitted is determined by thres, for which we do not provide a definition. Finally, remDiffTag removes differently-tagged HMaps from each group, and the groups are merged via mergeAliases as before.

3.2.4 Implementation

Further passes are used in the implementation. In particular, we trim unreachable aliases and remove aliases that simply point to another alias (like a_2 in mergeAliases) between each pass.

$$
\begin{aligned}
&\mathrm{req} : \Delta, a \longrightarrow m\\
&\mathrm{req}(\Delta, a) = \mathrm{req}(\Delta, a)\\
&\mathrm{req}(\Delta, (\mathrm{HMap}^{m}_{o})) = m
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{squashHorizontally} : \Delta \longrightarrow \Delta\\
&\mathrm{squashHorizontally}(\Delta) = \mathrm{fold}(\mathrm{mergeAliases}, \Delta, \mathrm{groupSimilarReq}(\Delta))
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{squashGlobal} : \Delta \longrightarrow \Delta\\
&\mathrm{squashGlobal} = \mathrm{squashHorizontally} \circ \mathrm{aliasSingleHMap}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{aliasSingleHMap} : \Delta \longrightarrow \Delta\\
&\mathrm{aliasSingleHMap}(\Delta) = \mathrm{fold}(\mathrm{f}, \Delta', \Delta'[A])\\
&\quad \text{where} \quad \Delta' = \mathrm{fold}(\mathrm{singleHMap}, \Delta, \Delta[\Gamma])\\
&\qquad\qquad \mathrm{f}(\Delta_0, \tau(\overline{\sigma}^{n})) = (\Delta_n, \tau(\sigma')), \text{ if } \tau = (\mathrm{HMap}^{m}_{o})\\
&\qquad\qquad\quad \text{where } \overline{(\Delta_i, \sigma_i) = \mathrm{singleHMap}(\Delta_{i-1}, \sigma_i)}\\
&\qquad\qquad \mathrm{f}(\Delta, \tau) = \mathrm{singleHMap}(\Delta, \tau), \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{singleHMap} : \forall\alpha.\Delta, (\alpha, \tau) \longrightarrow \Delta\\
&\mathrm{singleHMap}(\Delta, (\mathrm{x}, \tau)) = \Delta[\mathrm{x} \mapsto \sigma]\\
&\quad \text{where} \quad (\Delta', \sigma) = \mathrm{postwalk}(\Delta, \tau, \mathrm{f})\\
&\qquad\qquad \mathrm{f}(\Delta, (\mathrm{HMap}^{m_1}_{m_2})) = \mathrm{reg}(\Delta, (\mathrm{HMap}^{m_1}_{m_2}))\\
&\qquad\qquad \mathrm{f}(\Delta, \tau) = (\Delta, \tau), \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{groupSimilarReq} : \Delta \longrightarrow \overline{\overline{a}}\\
&\mathrm{groupSimilarReq}(\Delta) = [\overline{a} | \overline{k} \in \mathrm{dom}(\mathrm{r}), \overline{a} = \mathrm{remDiffTag}(\mathrm{similarReq}(\overline{k}))]\\
&\quad \text{where} \quad \mathrm{r} = \{(\overline{k}, \overline{a}) | (\mathrm{HMap}^{\{\overline{k\ \tau}\}}_{o}) \in \mathrm{rng}(\Delta[A]), \overline{a} = \mathrm{matchingReq}(\overline{k})\}\\
&\qquad\qquad \mathrm{matchingReq}(\overline{k}) = [a | (a, (\mathrm{HMap}^{m}_{o})) \in \Delta]\\
&\qquad\qquad \mathrm{similarReq}(\overline{k}) = [a | \overline{k'}^{n} \subseteq \overline{k}^{m}, m - n \leq \mathrm{thres}(m), a \in \mathrm{r}[\overline{k'}]]\\
&\qquad\qquad \mathrm{remDiffTag}(\overline{a}) = [a' | a' \in \overline{a},\ \text{if } (k, k') \in \mathrm{req}(\Delta, a') \text{ and } \bigvee \overline{(k, k'') \in \mathrm{req}(\Delta, a)} \text{ then } \overline{k' = k''}]
\end{aligned}
$$

Figure 10. Definition of squashGlobal$(\Delta) = \Delta'$

4 Extensions

4.1 Space-efficient tracking

To reduce the overhead of runtime tracking, we can borrow the concept of "space-efficient" contract checking from the gradual typing literature [13]. Instead of tracking just one path at once, a space-efficient implementation of track threads through a set of paths. When a tracked value flows into another tracked position, we extract the unwrapped value, and then our new tracked value tracks the paths that is the set of the old paths with the new path.

To model this, we introduce a new kind of value $[e, \rho]_{\mathrm{c}}{}^{v}_{\overline{\pi}}$ that tracks old value v as new value $[e, \rho]_{\mathrm{c}}$ with the paths $\overline{\pi}$. Proxy expressions are introduced when tracking functions, where instead of just returning a new wrapped function, we return a proxy. We can think of function proxies as a normal function with some extra metadata, so we can reuse the existing semantics for function application—in fact we can support space-efficient function tracking just by extending **track**.

We present the extension in Figure 11. The first two **track** rules simply make inference results for each of the paths. The next rule says that a bare closure reduces to a proxy that tracks the domain and range of the closure with respect to the list of paths. Attached to the proxy is everything needed to extend it with more paths, which is the role of the final rule. It extracts the original closure from the proxy and creates a new proxy with updated paths via the previous rule.

$$v ::= \ldots \mid [\lambda x.e, \rho]_{\mathrm{c}}{}^{[\lambda x.e, \rho]_{\mathrm{c}}}_{\overline{\pi}} \qquad \text{Values}$$

$$
\begin{aligned}
&\mathrm{track}(n, \overline{\pi}) &&= n \,; \bigcup \overline{\{\mathsf{N}_{\pi}\}}\\
&\mathrm{track}(k, \overline{\pi}) &&= k \,; \bigcup \overline{\{\mathbf{K}_{\pi}\}}\\
&\mathrm{track}([\lambda x.e, \rho]_{\mathrm{c}}, \overline{\pi}) &&= [e', \rho]_{\mathrm{c}}{}^{[\lambda x.e, \rho]_{\mathrm{c}}}_{\overline{\pi}} \,; \{\}\\
& &&\quad \text{where } y \text{ is fresh,}\\
& &&\quad e' = \lambda y.(\mathbf{track}\ \ ((\lambda x.e)\ (\mathbf{track}\ y\ \overline{\pi :: [\mathbf{dom}]}))\\
& &&\qquad\qquad\qquad\quad \overline{\pi :: [\mathbf{rng}]})\\
&\mathrm{track}([e', \rho']_{\mathrm{c}}{}^{[\lambda x.e, \rho]_{\mathrm{c}}}_{\overline{\pi'}}, \overline{\pi}) &&= \mathrm{track}([\lambda x.e, \rho]_{\mathrm{c}}, \overline{\pi} \cup \overline{\pi'})
\end{aligned}
$$

Figure 11. Space-efficient tracking extensions (changes)

4.2 Lazy tracking

Building further on the extension of space-efficient functions, we apply a similar idea for tracking maps. In practice, eagerly walking data structures to gather inference results is expensive. Instead, waiting until a data structure is used and tracking its contents lazily can help ease this tradeoff, with the side-effect that fewer inference results are discovered.

Figure 12 extends our system with lazy maps. We add a new kind of value $\{\overline{k\ v}\}^{\{\overline{k'\ \{\overline{m\ \overline{\pi}}\}}\}}$ that wraps a map $\{\overline{k\ v}\}$ with tracking information. Keyword entries k' are associated with pairs of type information m with paths $\overline{\pi}$. The first **track** rule demonstrates how to create a lazily tracked map. We calculate the possibly tagged entries in our type information in advance, much like the equivalent rule in Figure 7, and store them for later use. Notice that non-keyword entries are

$$
\begin{array}{lll}
v ::= \ldots \mid \{\overline{k\ v}\}^{\overline{\{k\ \{m\ \overline{\pi}\}\}}} & & \text{Values} \\
\\
\text{track}(\{\overline{k\ k'}\ \overline{k''\ v}\}, \overline{\pi}) & = \{\overline{k\ k'}\ \overline{k''\ v}\}^{\{\overline{k\ \mathrm{t}}\ \overline{k''\ \mathrm{t}}\}} ; \{\} & \\
 & \quad \text{where } \mathrm{t} = \{\{\overline{k\ k'}\ \overline{k''\ ?}\}\ \overline{\pi}\} & \\
\text{track}(\{\overline{k\ v}\}^{\overline{\{k'\ \{m\ \overline{\pi'}\}\}}}, \overline{\pi}) & = \{\overline{k\ v}\}^{\overline{\{k'\ \{m\ (\overline{\pi\cup} \cup \overline{\pi'\cup})\}\}}} ; \{\} &
\end{array}
$$

$$
\begin{array}{ll}
\delta(\text{assoc}, \{\overline{k\ v}\}^{\{k'\ \mathrm{t}', \overline{k''\ \mathrm{t}}\}}, k', v') & = \{\overline{k\ v}\}[k' \mapsto v']^{\{\overline{k''\ \mathrm{t}}\}}; \{\} \\
\delta(\text{assoc}, \{\overline{k\ v}\}^{\{\overline{k''\ \mathrm{t}}\}}, k', v') & = \{\overline{k\ v}\}[k' \mapsto v']^{\{\overline{k''\ \mathrm{t}}\}}; \{\} \\
\delta(\text{get}, \{k\ v, \overline{k'\ v'}\}^{\{k\ \mathrm{t}, \overline{k''\ \mathrm{t}'}\}}, k) & = \text{track}(v, \overline{\pi}) \\
\quad \text{where } \overline{\pi} = \left[\pi :: [\mathbf{key}_m(k)] \mid (m, \overline{\pi}) \in \mathrm{t}, \pi \in \overline{\pi}\right] & \\
\delta(\text{get}, \{k\ v, \overline{k'\ v'}\}^{\{\overline{k''\ \mathrm{t}'}\}}, k) & = v \\
\delta(\text{dissoc}, \{k\ v, \overline{k'\ v'}\}^{\{k\ \mathrm{t}, \overline{k''\ \mathrm{t}'}\}}, k) & = \{\overline{k'\ v'}\}^{\{\overline{k''\ \mathrm{t}'}\}} ; \{\} \\
\delta(\text{dissoc}, \{k\ v, \overline{k'\ v'}\}^{\{\overline{k''\ \mathrm{t}'}\}}, k) & = \{\overline{k'\ v'}\}^{\{\overline{k''\ \mathrm{t}'}\}} ; \{\}
\end{array}
$$

Figure 12. Lazy tracking extensions (changes)

not yet traversed, and thus no inference results are derived from them. The second **track** rule adds new paths to track.

The subtleties of lazily tracking maps lie in the δ rules. The assoc and dissoc rules ensure we no longer track overwritten entries. Then, the get rules perform the tracking that was deferred from the **track** rule for maps in Figure 7 (if the entry is still tracked).

In our experience, some combination of lazy and eager tracking of maps strikes a good balance between performance overhead and quantity of inference results. Intuitively, if a function does not access parts of its argument, they should not contribute to that function's type signature. However, our inference algorithm combines information *across* function signatures to deduce useful, recursive type aliases. Some eager tracking helps normalize the quality of function annotations with respect to unit test coverage.

For example, say functions f and g operate on the same types of (deeply nested) arguments, and f has complete test coverage (but does not traverse all of its arguments), and g has incomplete test coverage (but fully traverses its arguments). Eagerly tracking f would give better inference results, but lazily tracking g is more efficient. Forcing several layers of tracking helps strike this balance, which our implementation exposes as a parameter.

This can be achieved in our formal system by adding fuel arguments to **track** that contain depth and breadth tracking limits, and defer to lazy tracking when out of fuel.

4.3 Automatic contracts with clojure.spec

While we originally designed our tool to generate Typed Clojure annotations, it also supports generating "specs" for clojure.spec, Clojure's runtime verification system. There are key similarities between Typed Clojure and clojure.spec, such as extensive support for potentially-tagged keyword maps, however spec features a global registry of names via **`s/def`** and an explicit way declare unions of maps with a common dispatch key in `s/multi-spec`. These require differences in both type and name generation.

The following generated specs correspond to the first `Op` case of Figure 5 (lines 2-8).

```
(defmulti op-multi-spec :op) ;dispatch on :op key
(defmethod op-multi-spec :binding ;match :binding
  [_] ;s/keys matches keyword maps
  (s/keys :req-un [::op ...] ;required keys
          :opt-un [::column ...])) ;optional keys
(s/def ::op #{:js :let ...}) ;;op key maps to keywords
(s/def ::column int?) ;:column key maps to ints
; register ::Op as union dispatching on :op entry
(s/def ::Op (s/multi-spec op-multi-spec :op))
; emit's first argument :ast has spec ::Op
(s/fdef emit :args (s/cat :ast ::Op) :ret nil?)
```

5 Evaluation

We performed a quantitative evaluation of our algorithm on several open source programs.

startrek-clojure A reimplementation of a Star Trek text adventure game, created as a way to learn Clojure.

math.combinatorics The core library for common combinatorial functions on collections, with implementations based on Knuth's Art of Computer Programming, Volume 4.

fs A Clojure wrapper library over common file-system operations.

data.json A library for working with JSON.

mini.occ A model of occurrence typing by an author of the current paper. It utilizes three mutually recursive ad-hoc structures to represent expressions, types, and propositions.

cljs.compiler ClojureScript (CLJS) is a Clojure variant that runs on JavaScript virtual machines. We infer types for its compiler (written in Clojure) which emits JavaScript from a recursively defined map-based abstract syntax tree format.

Our approach is to carry out three experiments.

5.1 Experiment 1: Manual inspection

In the first experiment, we generate types for a program and manually inspect the resulting annotations. We follow the criteria outlined in Section 2 to judge the quality of our output, namely:

- using recognizable names,
- favoring compact annotations, and
- not overspecifying types.

Since it is our largest benchmark with 448 lines of generated type annotations, we concentrated on manually inspecting cljs.compiler's generated types. We commented (Section 2) on how successfully Figure 5 follows our goals.

Here, we further elaborate on Figure 5 (with line references), and contextualize them with the elided annotations.

One remarkable success in the generated types was the automatic inference Op (lines 1-12) with 14 distinct cases, and other features described in Figure 5. Further investigation reveals that the compiler actually features 36 distinct AST nodes—unsurprisingly, 39 assertions was not sufficient test coverage to discover them all. However, because of the recognizable name and organization of Op, it's clear where to add the missing nodes if no further tests are available.

A failure of cljs.compiler's generated types was HMap49305. It clearly fails to be a recognizable name. However, all is not lost: the compactness and recognizable names of other adjacent annotations makes it plausible for a programmer with some knowledge of the AST representation to recover. In particular 13/14 cases in Op have entries from :env to HMap49305, (like lines 9 and 10), and the only exception (line 7) maps to ColumnLineContextMap. From this information the user can decide to combine these aliases.

Several instances of overspecification are evident, such as the :statements entry of a :do AST node being inferred as an always-empty vector (line 11). In some ways, this is useful information, showing that test coverage for :do nodes could be improved. To fix the annotation, we could rerun the tool with better tests. If no such test exists, we would have to fall back to reverse-engineering code to identify the correct type of :statements, which is (**Vec** Op).

Finally, 19 functions in cljs.compiler are annotated to take or return Op (like lines 23, 24). This kind of alias reuse enables annotations to be relatively compact (only 16 type aliases are used by the 49 functions that were exercised).

5.2 Experiment 2: Changes needed to type check

In this experiment, we first generated types with our algorithm by running the tests, then amended the program so that it type checks. We observed some frequent reasons for why changes were needed (summarized in Figure 13).

Uncalled functions Some functions not called at all in the unit tests. This results in very general type annotations that need manual changes to be useful. For example, the startrek-clojure game has several exit conditions, one of which is running out of time. Since the tests do not specifically call this function, nor play the game long enough to invoke this condition, no useful type is inferred.

```
(ann game-over-out-of-time AnyFunction)
```

In this case, minimal effort is needed to amend this type signature: the appropriate type alias already exists:

```
(defalias CurrentKlingonsCurrentSectorEnterpriseMap
  (HMap :mandatory
    {:current-klingons (Vec EnergySectorMap),
     :current-sector (Vec Int), ...}
    :optional {:lrs-history (Vec Str)}))
```

So we amend the signature as

```
(ann game-over-out-of-time
  [(Atom1 CurrentKlingonsCurrentSectorEnterpriseMap)
   -> Boolean])
```

Over-precision Returns of function types are often too restrictive, as the unit tests may have not been complete.

There are several instances of this in math.combinatorics. The all-different? function takes a collection and returns true only if the collection contains distinct elements. As evidenced in the generated type, the tests exercise this functions with collections of integers, atoms, keywords, and characters.

```
(ann all-different?
  [(Coll (U Int (Atom1 Int) ':a ':b Character))
   -> Boolean])
```

In our experience, the union is very rarely a good candidate for a Typed Clojure type signature, so a useful heuristic to improve the generated types would be to upcast such unions to a more permissive type, like **Any**. When we performed that case study, we did not yet add that heuristic to our tool, so in this case, we manually amend the signature as

```
(ann all-different? [(Coll Any) -> Boolean])
```

Another example of overprecision is the generated type of initial-perm-numbers a helper function taking a *frequency map*—a hash map from values to the number of times they occur—which is the shape of the return value of the core frequencies function.

The generated type shows only a frequency map where the values are integers are exercised.

```
(ann initial-perm-numbers
  [(Map Int Int) -> (Coll Int)])
```

A more appropriate type instead takes (**Map Any Int**). In many examples of overprecision, while the generated type might not be immediately useful to check programs, they serve as valuable starting points and also provide an interesting summary of test coverage.

Missing polymorphism We do not attempt to infer polymorphic function types, so these amendments are expected. However, it is useful to compare the optimal types with our generated ones.

For example, the remove-nth function in math.combinatorics returns a functional delete operation on its argument. Here we can see the tests only exercise this function with collections of integers.

```
(ann remove-nth [(Coll Int) Int -> (Vec Int)])
```

However, the overall shape of the function is intact, and the manually amended type only requires a few keystrokes.

```
(ann remove-nth
  (All [a] [(Coll a) Int -> (Vec a)]))
```

Lib	LOC	GT	LA	MD	C	I	P	L	S	O	U	N	V	R	K	F	H	LS	RS	IT	MS	UT
sc	166	133	3	70/41	5	0	0	2	13	1	5	1	1	2	0	0	0	25	0	10	0	Y
mc	923	395	147	124/120	23	1	11	19	2	5	0	9	3	2	4	1	3	601	0	320	0	Y
fs	588	157	1	119/86	50	0	0	2	3	4	4	11	2	9	0	0	0	543	0	215	0	Y
dj	528	168	9	94/125																		
mo	530	49	1	46/26																		
cc	1776	448	4	N/A																		

Figure 13. *The number of type annotations generated for each program*: Lib = Abbreviated library names in the order we introduce them on page 9, LOC = Number of lines of code we generate types for, GT = Total number of lines of generated types after running our tool, LA = The number of local annotations generated by our tools. *Number of manual changes needed to type check, and why they were needed*: MD = Lines added/removed diff from git comparing initial generated types to the manual amendments needed to type check with Typed Clojure (unless it was too difficult to port), C = Casts, I = Instantiation, P = Polymorphic annotation, L = Local annotation, S = Work around type system Shortcoming, O = Overprecise argument type, U = Uncalled function due to bad test coverage, N = Add No-check annotation to skip checking function, V = Add Variable arity argument type, R = Overprecise return type, K = Add Keyword argument types, F = Added filter annotation, H = Erase/upcast HVec annotation. *Generated specs*: LS = Number of lines of spec generated, RS = No. recursive specs, IT = No. instance testing specs, MS = Useful map types, UT = Passed unit tests with specs enabled.

Similarly, `iter-perm` could be polymorphic, but its type is generated as

```
(ann iter-perm [(Vec Int) -> (U nil (Vec Int))])
```

We decided this function actually works over any number, and bounded polymorphism was more appropriate, encoding the fact that the elements of the output collection are from the input collection.

```
(ann iter-perm
  (All [a]
    [(Vec (I a Num)) -> (U nil (Vec (I a Num)))]))
```

Missing argument counts Often, variable argument functions are given very precise types. Our algorithm does not apply any heuristics to approximate variable arguments — instead we emit types that reflect only the arities that were called during the unit tests.

A good example of this phemonenon is the type inferred for the `plus` helper function from `math.combinatorics`. From the generated type, we can see the tests exercise this function with 2, 6, and 7 arguments.

```
(ann plus (IFn [Int Int Int Int Int Int Int -> Int]
               [Int Int Int Int Int Int -> Int]
               [Int Int -> Int]))
```

Instead, `plus` is actually variadic and works over any number of arguments. It is better annotated as the following, which is easy to guess based on both the annotated type and manually viewing the function implementation.

```
(ann plus [Int * -> Int])
```

A similar issue occurs with `mult`.

```
(ann mult [Int Int -> Int]) ;; generated
(ann mult [Int * -> Int])   ;; amended
```

A similar issue is inferring keyword arguments. Clojure implements keyword arguments with normal variadic arguments. Notice the generated type for `lex-partitions-H`, which takes a fixed argument, followed by some optional integer keyword arguments.

```
(ann lex-partitions-H
  (IFn [Int -> (Coll (Coll (Vec Int)))]
       [Int ':min Int ':max Int
        -> (Coll (Coll (Coll Int)))]))
```

While the arity of the generated type is too specific, we can conceivably use the type to help us write a better one.

```
(ann lex-partitions-H
  [Int & :optional {:min Int :max Int}
   -> (Coll (Coll (Coll Int)))])
```

Weaknesses in Typed Clojure We encountered several known weaknesses in Typed Clojure's type system that we worked around. The most invasive change needed was in startrek-clojure, which strongly updated the global mutable configuration map on initial play . We instead initialized the map with a dummy value when it is first created.

cljs.compiler uses many polymorphic idioms that Typed Clojure is poor at checking, so we deemed it too difficult to attempt to type check. In particular, there are many of usages of the core functions **`get-in`** and **`update-in`** (functions that deeply lookup and manipulate maps) which are not even assigned types in Typed Clojure. Many function definitions would need to be ignored by the type checker to work around this. Furthermore, many manual instantiations would be needed to check transducers and polymorphic functions passed to other polymorphic functions.

5.3 Experiment 3: Specs pass unit tests

Our final experiment uses our tool to generate specs (4.3) instead of types. Specs are checked at runtime, so to verify the utility of generated specs, we enable spec checking while rerunning the unit tests that were used in the process of creating them.

At first this might seem like a trivial property, but it serves as a valuable test of our inference algorithm. The aggressive merging strategies to minimize aliases and maximize recognizability, while unsound transformations, are based on hypotheses about Clojure idioms and how Clojure programs are constructed. If, hypothetically, we generated singleton specs for numbers like we do for keywords and did not eventually upcast them to **`number?`**, the specs might be too strict to pass its unit tests. Some function specs also perform generative testing based on the argument and return types provided. If we collapse a spec too much and include it in such a spec, it might feed a function invalid input.

Thankfully, we avoid such pitfalls, and so our generated specs pass their tests for the benchmarks we tried. The right of Figure 13 shows our preliminary results. All inferred specs pass the unit tests when enforced, which tells us they are at least well formed. Since hundreds of invariants are checked, we can also be more confident that the specs are useful.

6 Related work

Automatic annotations There are two common implementation strategies for automatic annotation 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 uses runtime analysis results to build up invariant/type knowledge.

Examples of invariant detection tools include Daikon [8], DIDUCE [11], and Carrot [19], and typically enhance statically typed languages with more expressive types or contracts. Examples of dynamic type inference include our tool, Rubydust [4], JSTrace [20], and TypeDevil [18], and typically target untyped languages.

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 know `x : Bottom`, 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 ∪ Integer` in our set of known invariants.

Daikon 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 [22], and gives the option to manually annotate necessary dependencies in function signatures when needed.

Other Annotation Tools Static analyzers TSInfer [15], Typpete [12] and Pytype [10] automatically annotate JavaScript and Python code, respectively. We were unable to install TSInfer, and unable to run Typpete without a runtime error. Pytype inferred `nodes` in Figure 1 as `(? -> int)`—our tool generates a compact recursive type (Figure 3). It inferred a class-based translation of Figure 1 similarly—fields `left`, `right`, and `val` were inferred as `Any`, and method `nodes` as `(self -> int)` on `Leaf`, and `(self -> Any)` on `Node`.

NoRegrets [17] uses dynamic analysis to learn how a program is used, and automatically runs the tests of downstream projects to improve test coverage. Their *dynamic access paths* represented as a series of *actions* are analogous to our paths of path elements.

7 Conclusion

This paper shows how to generate recursive heterogeneous type annotations for untyped programs that use plain data. We use a novel algorithm to "squash" the observed structure of program values into named recursive types suitable for optional type systems, all without the assistance of record, structure, or class definitions. We test this approach on thousands of lines of Clojure code, optimizing generated annotations for programmer comprehensibility over soundness.

In our experience, our guidelines to automatically name, group, and reuse types yield insightful annotations for those with some familiarity with the original programs, even if the initial annotations are imprecise, incomplete, and always require some changes to type check. Most importantly, many of these changes will involve simply rearranging or changing parts of existing annotations, so programmers are no longer left alone with the daunting task of reverse-engineering such programs completely from scratch.

References

[1] 2018. *Flow*. https://flow.org

[2] 2018. *Hack Language*. https://hacklang.org

[3] 2018. *TypeScript*. https://www.typescriptlang.org

[4] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. 2011. Dynamic Inference of Static Types for Ruby. *SIGPLAN Not.* 46, 1 (Jan. 2011), 459–472. https://doi.org/10.1145/1925844.1926437

[5] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. 2016. Practical Optional Types for Clojure. In *European Symposium on Programming Languages and Systems*. Springer, 68–94.

[6] Gilad Bracha. 2004. Pluggable type systems. In *OOPSLA workshop on revival of dynamic languages*, Vol. 4.

[7] Dropbox. [n. d.]. *PyAnnotate*. https://github.com/dropbox/pyannotate

[8] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123.

[9] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *ACM SIGPLAN Notices*, Vol. 37. ACM, 48–59.

[10] Google. [n. d.]. *PyType*. https://github.com/google/pytype

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

[12] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-Based Type Inference for Python 3. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 12–19.

[13] David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient Gradual Typing. *Higher Order Symbol. Comput.* 23, 2 (June 2010), 167–189. https://doi.org/10.1007/s10990-011-9066-z

[14] Rich Hickey. 2008. The Clojure programming language. In *Proc. DLS*.

[15] Erik Krogh Kristensen and Anders Møller. 2017. Inference and Evolution of TypeScript Declaration Files. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 99–115.

[16] Jukka Lehtosalo. [n. d.]. *mypy*. http://mypy-lang.org/

[17] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type Regression Testing to Detect Breaking Changes in Node.js Libraries. In *Proc. 32nd European Conference on Object-Oriented Programming (ECOOP)*.

[18] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 314–324.

[19] Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P Reiss. 2003. Automated fault localization using potential invariants. *arXiv preprint cs/0310040* (2003).

[20] Claudiu Saftoiu. 2010. *JSTrace: Run-time type discovery for JavaScript*. Technical Report. Technical Report CS-10-05, Brown University.

[21] Uri Shaked. 2018. *TypeWiz*. https://github.com/urish/typewiz

[22] Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical Types for Untyped Languages. In *Proc. ICFP (ICFP '10)*.