Squash the work

A Workflow for Typing Untyped Programs that use Ad-Hoc Data Structures

AMBROSE BONNAIRE-SERGEANT, SAM TOBIN-HOCHSTADT, Indiana University, USA

We present a semi-automated workflow for porting untyped programs to annotation-driven optional type systems. Unlike previous work, we infer useful types for recursive heterogeneous entities that have "ad-hoc" representations as plain data structures like maps, vectors, and sequences.

Our workflow starts by using dynamic analysis to collect samples from program execution via test suites or examples. Then, initial type annotations are inferred by combining observations across different parts of the program. Finally, the programmer uses the type system as a feedback loop to tweak the provided annotations until they type check.

Since inferring perfect annotations is usually undecidable and dynamic analysis is necessarily incomplete, the key to our approach is generating close-enough annotations that are easy to manipulate to their final form by following static type error messages. We explain our philosophy behind achieving this along with a formal model of the automated stages of our workflow, featuring maps as the primary "ad-hoc" data representation.

We report on using our workflow to convert real untyped Clojure programs to type check with Typed Clojure, which both feature extensive support for ad-hoc data representations. First, we visually inspect the initial annotations for conformance to our philosophy. Second, we quantify the kinds of manual changes needed to amend them. Third, we verify the initial annotations are meaningfully underprecise by enforcing them at runtime.

We find that the kinds of changes needed are usually straightforward operations on the initial annotations, leading to a substantial reduction in the effort required to port such programs.

1 INTRODUCTION

Consider the exercise of counting binary tree nodes using JavaScript. With a class-based tree representation, we naturally add a method to each kind of node like so.

```
class Node { nodes() { return 1 + this.left.nodes() + this.right.nodes(); } }
class Leaf { nodes() { return 1; } }
new Node(new Leaf(1), new Leaf(2)).nodes(); //=> 3 (constructors implicit)
```

An alternative "ad-hoc" representation uses plain JavaScript Objects with explicit tags. Then, the method becomes a recursive function that explicitly takes a tree as input. We trade the extensiblility and (presumably) speed of a method for a simple, reversible serialization to JSON.

```
function nodes(t) { switch t.op {
                      case "node": return 1 + nodes(t.left) + nodes(t.right);
                      case "leaf": return 1; } }
nodes({op: "node", left:{op: "leaf", val: 1}, right:{op: "leaf", val: 2}})//=>3
```

Now, consider the problem of inferring type annotations for these programs. The class-based representation is idiomatic to popular dynamic languages like JavaScript and Python, and so many existing solutions support it. For example, TypeWiz [Shaked 2018] uses dynamic analysis to generate the following TypeScript annotations from the above example execution of `nodes`.

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

The intuition behind inferring such a type is straightforward. For example, an instance of `Leaf` was observed in `Node`'s `left` field, and so the nominal type `Leaf` is used for its annotation.

The second “ad-hoc” style of programming seems peculiar in JavaScript, Python, and, indeed, object-oriented style in general. Correspondingly, existing state-of-the-art automatic annotation tools are not designed to support them. There are several ways to trivially handle such cases. Some enumerate the tree representation “verbatim” in a union, like TypeWiz [Shaked 2018].

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

Others “discard” most (or all) structure, like Typette [Hassan et al. 2018] and PyType [Google 2018] for Python.

```
def nodes(t: Dict[(Sequence, object)]) -> int: ... # Typette
def nodes(t) -> int: ...                           # PyType
```

Each annotation is clearly insufficient to meaningfully check both the function definition and valid usages. To show a desirable annotation for the “ad-hoc” program, we port it to Clojure [Hickey 2008], where it enjoys full support from the built-in runtime verification library clojure.spec and primary optional type system Typed Clojure [Bonnaire-Sergeant et al. 2016].

```
(defn nodes [t] (case (:op t)
                  :node (+ 1 (nodes (:left t)) (nodes (:right t)))
                  :leaf 1))
(nodes {:op :node, :left {:op :leaf, :val 1}, :right {:op :leaf, :val 2}}) ;=>3
```

Making this style viable requires a harmony of language features, in particular to support programming with functions and immutable values, but none of which comes at the expense of object-orientation. Clojure is hosted on the Java Virtual Machine and has full interoperability with Java objects and classes—even Clojure’s core design embraces object-orientation by exposing a collection of Java interfaces to create new kinds of data structures. The `{k v ...}` syntax creates a persistent and immutable Hash Array Mapped Trie [Bagwell 2001], which can be efficiently manipulated by dozens of built-in functions. The leading colon syntax like `:op` creates an interned *keyword*, which are ideal for map keys for their fast equality checks, and also look themselves up in maps when used as functions (e.g., `(:op t)` is like JavaScript’s `t.op`). *Multimethods* regain the extensibility we lost when abandoning methods, like the following.

```
(defmulti nodes-mm :op)
(defmethod nodes-mm :node [t] (+ 1 (nodes-mm (:left t)) (nodes-mm (:right t))))
(defmethod nodes-mm :leaf [t] 1)
```

On the type system side, Typed Clojure supports a variety of heterogeneous types, in particular for maps, along with occurrence typing [Tobin-Hochstadt and Felleisen 2010] to follow local control flow. Many key features come together to represent our “ad-hoc” binary tree as the following type.

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

The **`defalias`** form introduces an equi-recursive type alias `Tree`, **`U`** a union type, `'{:kw Type ...}` for heterogeneous keyword map types, and `':node` for keyword singleton types. With the following function annotation, Typed Clojure can intelligently type check the definition and usages of `nodes`.

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

This (manually written) Typed Clojure annotation involving `Tree` is significantly different from TypeWiz's "verbatim" annotation for `nodes`. First, it is recursive, and so supports trees of arbitrary depth (TypeWiz's annotation supports trees of height < 3). Second, it uses singleton types `':leaf` and `':node` to distinguish each case (TypeWiz upcasts `"leaf"` and `"node"` to `string`). Third, the tree type is factored out under a name to enhance readability and reusability. On the other end of the spectrum, the "discarding" annotations of Typette and PyType are too imprecise to use meaningfully (they include trees of arbitrary depth, but also many other values).

The challenge we overcome in this research is to automatically generate annotations like Typed Clojure's `Tree`, in such a way that the ease of manual amendment is only mildly reduced by unresolvable ambiguities and incomplete data collection.

2 OVERVIEW

We demonstrate our approach by synthesizing a Typed Clojure annotation for `nodes`. The following presentation is somewhat loose to keep from being bogged down by details—interested readers may follow the pointers to subsequent sections where they are made precise.

We use dynamic analysis to observe the execution of functions, so we give an explicit test suite for `nodes`.

```
(def t1 {:op :node, :left {:op :leaf, :val 1}, :right {:op :leaf, :val 2}})
(deftest nodes-test (is (= (nodes t1) 3)))
```

The first step is the instrumentation phase (formalized in Section 3.1), which monitors the inputs and outputs of `nodes` by redefining it to use the `track` function like so (where `<nodes-body>` begins the **`case`** expression of the original `nodes` definition):

```
(def nodes (fn [t'] (track ((fn [t] <nodes-body>) (track t' ['nodes :dom]))
                           ['nodes :rng])))
```

The `track` function (given later in Figure 2) takes a value to track and a *path* that represents its origin, and returns an instrumented value along with recording some runtime samples about the value. A path is represented as a vector of *path elements*, and describes the source of the value in question. For example, `(track 3 ['nodes :rng])` returns `3` and records the sample **`Int`**`['nodes :rng]` which says "**`Int`** was recorded at `nodes`'s range." Running our test suite `nodes-test` with an instrumented `nodes` results in more samples like this, most which use the path element `{:key :kw}` which represents a map lookup on the `:kw` entry.

```
':leaf['nodes :dom {:key :op}]      ':node['nodes :dom {:key :op}]      ?['nodes :dom {:key :val}]
                ?['nodes :dom {:key :left}]      ?['nodes :dom {:key :right}]
  ':leaf['nodes :dom {:key :left} {:key :op}]     ':leaf['nodes :dom {:key :right} {:key :op}]
    Int['nodes :dom {:key :left} {:key :val}]      Int['nodes :dom {:key :right} {:key :val}]
```

Now, our task is to transform these samples into a readable and useful annotation. This is the function of the inference phase (formalized in Section 3.2), which is split into three passes: first it generates a naive type from samples, then it combines types that occur syntactically near eachother ("squash locally"), and then aggressively across different function annotations ("squash globally").

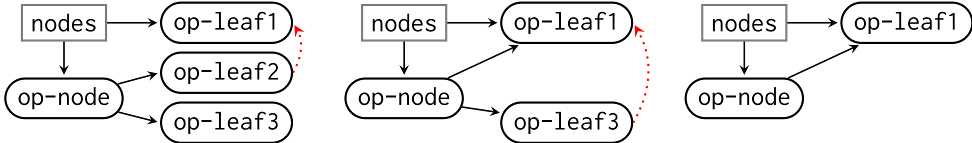
The initial naive type generated from these samples resembles TypeWiz's "verbatim" annotation given in Section 1, except the ? placeholder represents incomplete information about a path (this process is formalized as toEnv in Figure 3).

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

148 Next, the two "squashing" phases. The intuition behind both are based on seeing types as directed
149 graphs, where vertices are type aliases, and an edge connects two vertices u and v if u is mentioned
150 in v's type.
151 Local squashing (squashLocal in Figure 4) constructs such a graph by creating type aliases
152 from map types using a post-order traversal of the original types. In this example, the previous
153 annotations become:

```
(defalias op-leaf1 '{:op ':leaf, :val ?})
(defalias op-leaf2 '{:op ':leaf, :val Int})
(defalias op-leaf3 '{:op ':leaf, :val Int})
(defalias op-node '{:op ':node, :left op-leaf2, :right op-leaf3})
(ann nodes [(U op-leaf1 op-node) -> Int])
```

160 As a graph, this becomes the left-most graph below. The dotted edge from `op-leaf2` to `op-leaf1`
161 signifies that they are to be merged, based on the similar structure of the types they point to.



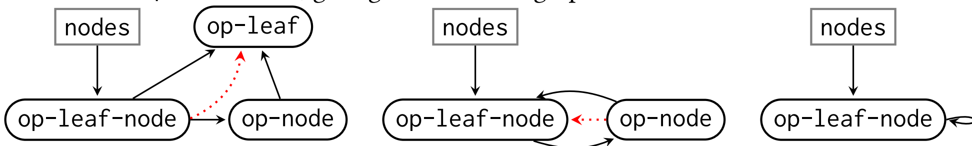
167 After several merges (reading the graphs left-to-right), local squashing results in the following:

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

172 All three duplications of the `':leaf` type in the naive annotation have been consolidated into
173 their own name, with the ? placeholder for the `:val` entry being absorbed into **`Int`**.
174 Now, the global squashing phase (squashGlobal in Figure 5) proceeds similarly, except the notion
175 of a vertex is expanded to also include *unions* of map types, calculated, again, with a post-order
176 traversal of the types giving:

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

182 This creates `op-leaf-node`, giving the left-most graph below.



187 Now, type aliases are merged based on overlapping *sets* of top-level keysets and likely tags. Since
188 `op-leaf` and `op-leaf-node` refer to maps with identical keysets (`:op` and `:val`) and whose likely
189 tags agree (the `:op` entry is probably a tag, and they are both `':leaf`), they are merged and all
190 occurrences of `op-leaf` are renamed to `op-leaf-node`, creating a *mutually recursive type* between
191 the remaining aliases in the middle graph:

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

In the right-most graph, the aliases op-node and op-leaf-node are merged for similar reasons:

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

All that remains is to choose a recognizable name for the alias. Since all its top-level types seem to use the :op entry for tags, we choose the name Op and output the final annotation:

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

The rest of the porting workflow involves the programmer repeatedly type checking their code and gradually tweaking the generated annotations until they type check. It turns out that this annotation immediately type checks the definition of nodes and all its valid usages, so we turn to a more complicated function visit-leaf to demonstrate a typical scenario.

```
(defn visit-leaf "Updates :leaf nodes in tree t with function f."
  [f t] (case (:op t)
          :node (assoc t :left (visit-leaf f (:left t))
                         :right (visit-leaf f (:right t)))
          :leaf (f t)))
```

This higher-order function uses **assoc** to associate new children as it recurses down a given tree to update leaf nodes with the provided function. The following test simply increments the leaf values of the previously-defined t1.

```
(deftest visit-leaf-test
  (is (= (visit-leaf (fn [leaf] (assoc leaf :val (inc (:val leaf)))) t1)
         {:op :node, :left {:op :leaf, :val 2}, :right {:op :leaf, :val 3}})))
```

Running this test under instrumentation yields some interesting runtime samples whose calculation is made efficient by space-efficient tracking (Section 4.1), which ensures a function is not repeatedly tracked unnecessarily. The following two samples demonstrate how to handle multiple arguments (by parameterizing the :dom path element) and higher-order functions (by nesting :dom or :rng path elements).

$\texttt{':leaf}_{\texttt{['visit-leaf \{:dom 1\} \{:key :op\}]}}$ $\texttt{':leaf}_{\texttt{['visit-leaf \{:dom 0\} \{:dom 0\} \{:key :op\}]}}$

Here is our automatically generated initial annotation.

```
(defalias Op (U '{:op ':leaf, :val t/Int} '{:op ':node, :left Op, :right Op}))
(ann visit-leaf [[Op -> Any] Op -> Any])
```

Notice the surprising occurrences of **Any**. They originate from ? placeholders due to the lazy tracking of maps (Section 4.2). Since visit-leaf does not traverse the results of f, nor does anything traverse visit-leaf's results (hash-codes are used for equality checking) neither tracking is realized. Also notice the first argument of visit-leaf is underprecise. These could trigger type errors on usages of visit-leaf, so manual intervention is needed (highlighted). We factor out and use a new alias Leaf and replace occurrences of **Any** with Op.

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

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

Fig. 1. Syntax of Terms, Types, Inference results, and Environments for λ_{track}

We measure the success of our workflow by using it to type check real Clojure programs. Experiment 1 (Section 5.1) manually inspects a selection of inferred types. Experiment 2 (Section 5.2) classifies and quantifies the kinds of changes needed. Experiment 3 (Section 5.3) enforces initial annotations at runtime to ensure they are meaningfully underprecise.

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:

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

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 1 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

295 value with respect to a sequence of path elements l, always starting with a variable x, and are read
296 left-to-right. Other path elements are a function domain **dom**, a function range **rng**, and a map
297 entry $\mathbf{key}_{\overline{k_1}}(k_2)$ which represents the result of looking up k_2 in a map with keyset $\overline{k_1}$.
298 Inference results $\{\overline{\tau_\pi}\}$ are pairs of paths π and types τ that say the path π was observed to
299 be type τ. Types τ are numbers N, function types $[\tau \rightarrow \tau]$, ad-hoc union types $(\bigcup\ \tau\ \tau)$, type
300 aliases a, and unknown type ? that represents a temporary lack of knowledge during the inference
301 process. Heterogeneous keyword map types $\{\overline{k\ \tau}\}$ for now represent a series of required keyword
302 entries—we will extend them to have optional entries in later phases.
303 The big-step operational semantics $\rho \vdash e \Downarrow v\ ;\ r$ (Figure 2) says under runtime environment ρ
304 expression e evaluates to value v with inference results r. Most rules are standard, with extensions
305 to correctly propagate inference results r. B-Track is the only interesting rule, which instruments
306 its fully-evaluated argument with the track metafunction.
307 The metafunction $\mathsf{track}(v, \pi) = v'\ ; r$ (Figure 2) says if value v occurs at path π, then return a
308 possibly-instrumented v' paired with inference results r that can be immediately derived from the
309 knowledge that v occurs at path π. It has a case for every kind of value. The first three cases records
310 the number input as type N. The fourth case, for closures, returns a wrapped value resembling
311 higher-order function contracts [Findler and Felleisen 2002], but we track the domain and range
312 rather than verify them. The remaining rules case, for maps, recursively tracks each map value,
313 and returns a map with possibly wrapped values. Immediately accessible inference results are
314 combined and returned. A specific rule for the empty map is needed because we otherwise only
315 rely on recursive calls to **track** to gather inference results—in the empty case, we have no data to
316 recur on.
317 Now we have sufficient pieces to describe the initial collection phase of our model. Given an
318 expression e and variables $\overline{x}$ to track, $\mathsf{instrument}(e, \overline{x}) = e'$ returns an instrumented expression e'
319 that tracked usages of $\overline{x}$. It is defined via capture-avoiding substitution:
320
321 $$\mathsf{instrument}(e, \overline{x}) = e[\overline{(\mathbf{track}\ x\ [x])/x}]$$
322 Then, the overall collection phase $\mathsf{collect}(e, \overline{x}) = r$ says, given an expression e and variables $\overline{x}$ to
323 track, returns inference results r that are the results of evaluating e with instrumented occurrences
324 of $\overline{x}$. It is defined as:
325
326 $$\mathsf{collect}(e, \overline{x}) = r, \text{ where } \vdash \mathsf{instrument}(e, \overline{x}) \Downarrow v\ ;\ r$$
327 For our running example of collecting for the program $(f\ \{\text{:a } 42\})$, we instrument the program
328 by wrapping occurrences of f with **track** with path $[f]$.
329
330 $$\mathsf{instrument}((f\ \{\text{:a } 42\}), [f]) = ((\mathbf{track}\ f\ [f])\ \{\text{:a } 42\})$$
331 Then we evaluate the instrumented program and derive two inference results (colored in red for
332 readability):
333 $$\vdash ((\mathbf{track}\ f\ [f])\ \{\text{:a } 42\}) \Downarrow 42\ ;\ \{\mathsf{N}_{[f, \mathbf{dom}, \mathbf{key}(\text{:a})]}, \mathsf{N}_{[f, \mathbf{rng}]}\}$$
334 Here is the full derivation:
335
```
336 => ((track f [f]) {:a 42})
337 => (track (get (track {:a 42} [f, dom]) :a) [f, rng])
338 => (track (get {:a 42} ; {N[f, dom, key(:a)]} :a) [f, rng])
339 => (track 42 ; {N[f, dom, key(:a)]} [f, rng])
340 => 42 ; {N[f, dom, key(:a)], N[f, rng]}
```
341 Notice that intermediate values can have inference results (colored) attached to them with a
342 semicolon, and the final value has inference results about both f's domain and range.
343

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

B-Clos
$$\rho \vdash \lambda x.e \Downarrow [\lambda x.e, \rho]_c \; ; \; \{\}$$

B-Val
$$\rho \vdash v \Downarrow v \; ; \; \{\}$$

B-Var
$$\rho \vdash x \Downarrow \rho(x) \; ; \; \{\}$$

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

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

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

Fig. 2. Operational semantics, $\text{track}(v, \pi) = v \; ; r$ and constants

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:

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

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 3. The entry point toEnv folds over inference results to create an initial type environment via update. This style

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

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

$$
\begin{array}{l}
\mathsf{fold} : \forall \alpha, \beta.(\alpha, \beta \to \alpha), \alpha, \overline{\beta} \to \alpha \\
\mathsf{fold}(\mathsf{f}, \mathsf{a}_0, \overline{\mathsf{b}}^{n}) = \mathsf{a}_n \\
\quad \text{where } \overline{\mathsf{a}_i = \mathsf{f}(\mathsf{a}_{i-1}, \mathsf{b}_i)}^{1 \le i \le n} \\
\\
\mathsf{toEnv} : r \to \Gamma \\
\mathsf{toEnv}(r) = \mathsf{fold}(\mathsf{update}, \{\}, r)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{update} : \Gamma, \tau_\pi \to \Gamma & & \\
\mathsf{update}(\Gamma, \tau_{\pi::[\mathbf{key}_{\{\overline{k'\ \sigma}\}}(k)]}) & = & \mathsf{update}(\Gamma, \{\overline{k'\ \sigma}\ k\ \tau\}_\pi) \\
\mathsf{update}(\Gamma, \tau_{\pi::[\mathbf{dom}]}) & = & \mathsf{update}(\Gamma, [\tau \to ?]_\pi) \\
\mathsf{update}(\Gamma, \tau_{\pi::[\mathbf{rng}]}) & = & \mathsf{update}(\Gamma, [? \to \tau]_\pi) \\
\mathsf{update}(\Gamma[x \mapsto \sigma], \tau_{[x]}) & = & \Gamma[x \mapsto \tau \sqcup \sigma] \\
\mathsf{update}(\Gamma, \tau_{[x]}) & = & \Gamma[x \mapsto \tau]
\end{array}
$$

Fig. 3. Definition of $\mathsf{toEnv}(r) = \Gamma$

is inspired by occurrence typing [Tobin-Hochstadt and Felleisen 2010], 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 2). This information helps us avoid prematurely collapsing tagged maps, by the side condition of the HMap $\sqcup$ case. The $\sqcup^{\mathsf{H}}$ 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 $[\mathsf{N} \to ?] \sqcup [\mathbf{K} \to ?] = [(\bigcup\ \mathsf{N}\ \mathbf{K}) \to ?]$.

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

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

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

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.

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

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

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

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

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

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

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

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

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

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

Fig. 4. Definition of squashLocal(Γ) = Δ

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

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

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

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

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

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

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

Fig. 5. Definition of squashGlobal$(\Delta) = \Delta'$

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 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

540 different arguments or functions) so that any deficiencies in unit testing coverage are massaged
541 away.
542 The squashGlobal function is the entry point in this pass, and is similar in structure to the
543 previous pass. It first creates aliases for each HMap via aliasSingleHMap. Then, HMap aliases are
544 grouped and merged in squashHorizontally.
545 The aliasSingleHMap function first traverses the type environment to create HMap aliases
546 via singleHMap, and binds the resulting envionment as Δ'. Then, alias environment entries are
547 updated with f, whose first case prevents re-aliasing a top-level HMap, before we call singleHMap
548 (singleHMap's second argument accepts both x and a). The $\tau(\overline{\sigma})$ syntax represents a type τ whose
549 constructor takes types $\overline{\sigma}$.
550 After that, squashHorizontally creates groups of related aliases with groupSimilarReq. Each
551 group contains HMap aliases whose required keysets are similar, but are never differently-tagged.
552 The code creates a map r from keysets to groups of HMap aliases with that (required) keyset. Then,
553 for every keyset $\overline{k}$, similarReq adds aliases to the group whose keysets are a subset of $\overline{k}$. The number
554 of missing keys permitted is determined by thres, for which we do not provide a definition. Finally,
555 remDiffTag removes differently-tagged HMaps from each group, and the groups are merged via
556 mergeAliases as before.
557
558 *3.2.4 Implementation.* Further passes are used in the implementation. In particular, we trim un-
559 reachable aliases and remove aliases that simply point to another alias (like a_2 in mergeAliases)
560 between each pass.
561
562 ## 4 EXTENSIONS

563 Two optimizations are crucial for practical implementations of the collection phase. First, space-
564 efficient tracking efficiently handles a common case with higher-order functions where the same
565 function is tracked at multiple paths. Second, instead of tracking a potentially large value by
566 eagerly traversing it, lazy tracking offers a pay-as-you-go model by wrapping a value and only
567 tracking subparts as they are accessed. Both were necessary to collect samples from the compiler
568 implementation we instrumented for Experiment 1 (Section 5.1) because it used many higher-order
569 functions and its AST representation can be quite large which made it intractible to eagerly traverse
570 each time it got passed to one of dozens of functions.
571
572 ### 4.1 Space-efficient tracking

573 To reduce the overhead of runtime tracking, we can borrow the concept of "space-efficient" contract
574 checking from the gradual typing literature [Herman et al. 2010]. Instead of tracking just one path
575 at once, a space-efficient implementation of track threads through a set of paths. When a tracked
576 value flows into another tracked position, we extract the unwrapped value, and then our new
577 tracked value tracks the paths that is the set of the old paths with the new path.
578 To model this, we introduce a new kind of value $[e, \rho]_{\mathrm{c}}\frac{v}{\overline{\pi}}$ that tracks old value v as new value
579 $[e, \rho]_{\mathrm{c}}$ with the paths $\overline{\pi}$. Proxy expressions are introduced when tracking functions, where instead
580 of just returning a new wrapped function, we return a proxy. We can think of function proxies as a
581 normal function with some extra metadata, so we can reuse the existing semantics for function
582 application—in fact we can support space-efficient function tracking just by extending **track**.
583 We present the extension in Figure 6. The first two **track** rules simply make inference results for
584 each of the paths. The next rule says that a bare closure reduces to a proxy that tracks the domain
585 and range of the closure with respect to the list of paths. Attached to the proxy is everything needed
586 to extend it with more paths, which is the role of the final rule. It extracts the original closure from
587 the proxy and creates a new proxy with updated paths via the previous rule.
588

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

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

Fig. 6. 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 7 extends our system with lazy maps. We add a new kind of value $\{\overline{k\ v}\}^{\overline{\{k'\ \{m\ \overline{\pi}\}\}}}$ 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 2, and store them for later use. Notice that non-keyword entries are 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 2 (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

$$v ::= \ldots \mid \{\overline{k\ v}\}^{\{\overline{k\ \{\overline{m\ \overline{\pi}}\}}\}} \qquad \text{Values}$$

$$\begin{aligned}
\mathrm{track}(\{\overline{k\ k'}\ \overline{k''\ v}\}, \overline{\pi}) &= \{\overline{k\ k'}\ \overline{k''\ v}\}^{\{\overline{k\ \mathrm{t}}\ \overline{k''\ \mathrm{t}}\}}\ ; \{\} \\
&\quad \text{where } \mathrm{t} = \{\overline{\{\overline{k\ k'}\ \overline{k''\ ?}\}\ \overline{\pi}}\} \\
\mathrm{track}(\{\overline{k\ v}\}^{\{\overline{k'\ \{\overline{m\ \overline{\pi'}}\}}\}}, \overline{\pi}) &= \{\overline{k\ v}\}^{\{\overline{k'\ \{m\ (\overline{\pi \cup}\ \cup\ \overline{\pi' \cup})\}}\}}\ ; \{\}
\end{aligned}$$

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

Fig. 7. Lazy tracking extensions (changes)

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 to 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 8 (lines 2-5).

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

5 EVALUATION

We performed a quantitative evaluation of our workflow on several open source programs in three experiments. We ported five programs to Typed Clojure with our workflow, and merely generated types for one larger program we deemed too difficult to port, but features interesting data types.

Experiment 1 involves a manual inspection of the types from our automatic algorithm. We detail our experience in generating types for part of an industrial-grade compiler which we ultimately decided not to manually port to Typed Clojure. This was because it uses many programming idioms beyond Typed Clojure's capabilities (those detailed as "Further Challenges" by Bonnaire-Sergeant et al. [2016]), and so the final part of the workflow mostly involves working around its shortcomings.

```
 1 (defalias Op ; omitted some entries and 11 cases
 2   (U (HMap :mandatory
 3            {:op ':binding, :info (U NameShadowMap FnScopeFnSelfNameNsMap), ...}
 4            :optional
 5            {:env ColumnLineContextMap, :init Op, :shadow (U nil Op), ...})
 6     '{:op ':const, :env HMap49305, ...}
 7     '{:op ':do, :env HMap49305, :ret Op, :statements (Vec Nothing), ...}
 8     ...))
 9 (defalias ColumnLineContextMap
10   (HMap :mandatory {:column Int, :line Int} :optional {:context ':expr}))
11 (defalias HMap49305 ; omitted some extries
12   (U nil
13      '{:context ':statement, :column Int, ...}
14      '{:context ':return, :column Int, ...}
15      (HMap :mandatory {:context ':expr, :column Int, ...} :optional {...})))
16 (ann emit [Op -> nil])
17 (ann emit-dot [Op -> nil])
```

Fig. 8. Sample generated types for cljs.compiler.

Experiment 2 studies the kinds of the manual changes needed to port our five programs to Typed Clojure, starting from the automatically generated annotations. Experiment 3 enforces the initially generated annotations for these programs at runtime to check they are meaningfully underprecise.

5.1 Experiment 1: Manual inspection

For the first experiment, we manually inspect the types automatically generated by our tool. We judge our tool's ability to use recognizable names, favor compact annotations, and not overspecify types.

We take this opportunity to juxtapose some strengths and weaknessess of our tool by discussing a somewhat problematic benchmark, a namespace from the ClojureScript compiler called cljs.compiler (the code generation phase). We generate 448 lines of type annotations for the 1,776 line file, and present a sample of our tool's output as Figure 8. We were unable to fully complete the porting to Typed Clojure due to type system limitations, but the annotations yielded by this benchmark are interesting nonetheless.

The compiler's AST format is inferred as `Op` (lines 1-8) with 22 recursive references (like lines 5, 5, 7) and 14 cases distinguished by `:op` (like lines 3, 6, 7), 5 of which have optional entries (like lines 4-5). 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), in part because of key optimizations discussed in Section 4.

The main function of the code generation phase 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 16 with a recursive AST type `Op` (lines 1-8).

Similar to our opening example `nodes`, it uses the `:op` key to disambiguate between (16) cases, and has recursive references (`Op`). We just present the first 4 cases. The first case `':binding` has 4 required and 8 optional entries, whose `:info` and `:env` entries refer to other **HMap** type aliases generated by the tool.

An important question to address is "how accurate are these annotations?". Unlike previous work in this area [An et al. 2011], 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 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 1-8 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 1-8) 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 9-10) enumerates the keys of the map type it points to,
- `NameShadowMap` and `FnScopeFnSelfNameNsMap` (line 3) similarly, and
- `HMap49305` (lines 11-15) shows how our tool fails to give names to certain combinations of types (we now discuss the severity of this particular situation).

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 6 and 7), and the only exception (line 5) maps to `ColumnLineContextMap`. From this information the user can decide to combine these aliases.

Favor compact annotations. Literally translating runtime observations into annotations without compacting them leads to unmaintainable and impractical types resembling TypeWiz's "verbatim" annotation for `nodes`. To avoid this, we use optional keys where possible, like line 10, infer recursive types like `Op`, and reuse type aliases in function annotations, like `emit` and `emit-dot` (lines 16, 17).

One remarkable success in the generated types was the automatic inference `Op` (lines 1-8) with 14 distinct cases, and other features described in Figure 8. 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.

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` is a good example of this—our inferred type 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 17) 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` (the dot operator in Clojure handles host interoperability) 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).

Several instances of overspecification are evident, such as the `:statements` entry of a `:do` AST node being inferred as an always-empty vector (line 7). 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 16, 17). 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

We used our workflow to port the following open source Clojure programs to Typed Clojure.

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.

In this experiment, we first generated types with our algorithm by running the tests, then amended the program so that it type checks. Figure 9 summarizes our results. After the lines of code we generate types for, the next two columns show how many lines of types were generated and the lines manually changed, respectively. The latter is a git line diff between commits of the initial generated types and the final manually amended annotations. While an objectively fair measurement, it is not a good indication of the effort needed to port annotations (a 1 character changes on a line is represented by 1 line addition and 1 line deletion) The rest of the table enumerates the different kinds of changes needed and their frequency.

Uncalled functions. A function without tests receives a broad type annotation that must be amended. 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:

Library	Lines of code	Lines of Generated Global/Local Types	Lines manually added/removed	Casts/Instantiations	Polymorphic annotation	Local annotation	Type System Workaround/no-check	Overprecise argument/return type	Uncalled function (bad test coverage)	Variable-arity/keyword arg type	Add occurrence typing annotation	Erase or upcast HVec annotation	Add missing case in defalias
startrek	166	133/3	70/41	5 / 0	0	2	13/1	1 /2	5	1 / 0	0	0	0
math.comb	923	395/147	124/120	23 / 1	11	19	2 /9	5 /2	0	3 / 4	1	3	0
fs	588	157/1	119/86	50 / 0	0	2	3 /11	4 /9	4	2 / 0	0	0	0
data.json	528	168/9	94/125	6 / 0	0	2	4 /5	11/7	5	0 / 20	0	0	0
mini.occ	530	49/1	46/26	7 / 0	0	2	5 /2	4 /2	6	0 / 0	0	1	5

Fig. 9. Lines of generated annotations, git line diff for total manual changes to type check the program, and the kinds of manual changes.

```
(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. Function types are often too restrictive due to insufficient unit tests.

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)]))
```

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.

Missing **`defalias`** *cases.* With insufficient test coverage, our tool can miss cases in a recursively defined type. In particular, mini.occ features three recursive types—for the representation of types `T`, propositions `P`, and expressions `E`. For `T`, three cases were missing, along with having to upcast the `:params` entry from the singleton vector `'[NameTypeMap]`. Two cases were missing from `E`. The manual changes are highlighted (`P` required no changes with five cases).

```
(defalias T
  (U '{:T ':not, :type T}
     '{:T ':refine, :name t/Sym, :prop P}
     '{:T ':union, :types (t/Set T)}
     '{:T ':false}
     '{:T ':fun,
       :params (t/Vec NameTypeMap),
       :return T}
     '{:T ':intersection, :types (Set T)}
     '{:T ':num}))
```

```
(defalias E
  (U '{:E ':add1}
     '{:E ':n?}
     '{:E ':app, :args (Vec E), :fun E}
     '{:E ':false}
     '{:E ':if, :else E, :test E, :then E}
     '{:E ':lambda, :arg Sym, :arg-type T,
       :body E}
     '{:E ':var, :name Sym}))
```

5.3 Experiment 3: Specs pass unit tests

Our final experiment uses our tool to generate specs (Section 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,

Library	LOC	Lines of specs	Recursive	Instance	Het. Map	Passed Tests?
startrek	166	25	0	10	0	Yes
math.comb	923	601	0	320	0	Yes
fs	588	543	0	215	0	Yes
data.json	528	401	0	174	0	No (1/79 failed)
mini.occ	530	131	3	25	15	Yes

Fig. 10. Summary of the quantity and kinds of generated specs and whether they passed unit tests when enabled. The one failing test was related to pretty-printing JSON, and seems to be an artifact of our testing environment, as it still fails with all specs removed.

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. Figure 10 shows our preliminary results. All inferred specs pass the unit tests when enforced, which tells us they are at least well formed. We had some seemingly unrelated difficulty with a test in data.json which we explain in the caption. Since hundreds of invariants are checked—mostly "instance" checks that a value is of a particular class or interface—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 [Ernst et al. 2001], DIDUCE [Hangal and Lam 2002], and Carrot [Pytlik et al. 2003], and typically enhance statically typed languages with more expressive types or contracts. Examples of dynamic type inference include our tool, Rubydust [An et al. 2011], JSTrace [Saftoiu 2010], and TypeDevil [Pradel et al. 2015], 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 [Tobin-Hochstadt and Felleisen 2010], and gives the option to manually annotate necessary dependencies in function signatures when needed.

Reverse Engineering Programs with Static Analysis. Rigi [Müller et al. 1992] analyzes the structure of large software systems, combining static analysis with a user-facing graphical environment to allow users to view and manipulate the in-progress reverse engineering results. We instead use a static type system as a feedback mechanism, which forces more aggressive compacting of generated annotations.

Lackwit [O'Callahan and Jackson 1997] uses static analysis to identify abstract data types in C programs. Like our work, they share representations between values, except they use type inference with representations encoded as types. Recursive representations are inferred via Felice and Coppos's work on type inference with recursive types [Cardone and Coppo 1991], where we rely on our imprecise "squashing" algorithms over incomplete runtime samples.

Soft Typing [Cartwright and Fagan 1991] uses static analysis to insert runtime checks into untyped programs for invariants that cannot be proved statically. Our approach is instead to let the user check the generated annotations with a static type system, with static type errors guiding the user to manually add casts when needed.

Schema Inference. Baazizi et al. [2017] infer structural properties of JSON data using a custom JSON schema format. Their schema inference algorithm proceeds in two stages: schema inference and schema fusion. This resembles our collection and naive type environment construction phases. There are slight differences between schema fusion and our approach. Schema fusion upcasts heterogeneous array types to be homogeneous, where we maintain heterogeneous vector types until a differently-sized vector type is found in the same position. We also support function types, which JSON lacks. While they support nested data, they do not attempt to factor out common types as names or create recursive types like our squashing algorithms.

DiScala and Abadi [2016] present a machine learning algorithm to translate denormalized and nested data that is commonly found in NoSQL databases to traditional relational formats used by standard RDBMS. A key component is a schema generation algorithm which arranges related data into tables via a matching algorithm which discovers related attributes. Phases 1 and 2 of their algorithm are similar to our local and global squashing algorithms, respectively, in that first locally accessible information is combined, and then global information. They identify groups of attributes that have (possibly cyclic) relationships. Where our squashing algorithms for map types are based on (sets of) keysets—on the assumption that related entities use similar keysets—they also join attributes based on their similar values. This enables more effective entity matching via equivalent attributes with different names (e.g., "Email" vs "UserEmail"). Our approach instead assumes programs are somewhat internally consistent, and instead optimizes to handle missing samples from incomplete dynamic analysis.

Other Annotation Tools. Static analyzers for JavaScript (TSInfer [Kristensen and Møller 2017]) and for Python (Typpete [Hassan et al. 2018] and PyType [Google 2018]) automatically annotate code

with types. PyType and Typpete inferred `nodes` as `(? -> int)` and `Dict[(Sequence, object)] -> int`, respectively—our tool infers it as `[Op ->Int]` by also generating a compact recursive type. Similarly, a class-based translation of inferred both `left` and `right` fields as `Any` by PyType, and as `Leaf` by Typpete—our tool uses `Op`, a compact recursive type containing *both* `Leaf` and `Node`. This is similar to our experience with TypeWiz in Section 1. (We were unable to install TSInfer.)

NoRegrets [Mezzetti et al. 2018] 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

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

Mohamed-Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Schema inference for massive JSON datasets. In *Extending Database Technology (EDBT)*.

Phil Bagwell. 2001. Ideal hash trees. *Es Grands Champs* 1195 (2001), 5–2.

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.

Felice Cardone and Mario Coppo. 1991. Type inference with recursive types: Syntax and semantics. *Information and Computation* 92, 1 (1991), 48–80.

Robert Cartwright and Mike Fagan. 1991. Soft Typing. In *Proc. PLDI.*

Michael DiScala and Daniel J Abadi. 2016. Automatic generation of normalized relational schemas from nested key-value data. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 295–310.

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.

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

Google. 2018. *PyType.* https://github.com/google/pytype

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.

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.

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

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

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.

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)*.

Hausi A Müller, Scott R Tilley, Mehmet A Orgun, BD Corrie, and Nazim H Madhavji. 1992. A reverse engineering environment based on spatial and visual software interconnection models. In *ACM SIGSOFT Software Engineering Notes*, Vol. 17. ACM, 88–98.

Robert O'Callahan and Daniel Jackson. 1997. Lackwit: A program understanding tool based on type inference. In *In Proceedings of the 19th International Conference on Software Engineering*. Citeseer.

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.

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

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

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

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