Fast and Precise Type Checking for JavaScript

AVIK CHAUDHURI, Facebook Inc., USA
PANAGIOTIS VEKRIS, University of California, San Diego, USA
SAM GOLDMAN, Facebook Inc., USA
MARSHALL ROCH, Facebook Inc., USA
GABRIEL LEVI, Facebook Inc., USA

In this paper we present the design and implementation of Flow, a fast and precise type checker for JavaScript that is used by thousands of developers on millions of lines of code at Facebook every day. Flow uses sophisticated type inference to understand common JavaScript idioms precisely. This helps it find non-trivial bugs in code and provide code intelligence to editors without requiring significant rewriting or annotations from the developer. We formalize an important fragment of Flow's analysis and prove its soundness. Furthermore, Flow uses aggressive parallelization and incrementalization to deliver near-instantaneous response times. This helps it avoid introducing any latency in the usual edit-refresh cycle of rapid JavaScript development. We describe the algorithms and systems infrastructure that we built to scale Flow's analysis.

CCS Concepts: • **Theory of computation → Type structures**; **Program analysis**;

Additional Key Words and Phrases: Type Systems, Type Inference, JavaScript

ACM Reference Format:
Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and Precise Type Checking for JavaScript. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 48 (October 2017), 30 pages.
https://doi.org/10.1145/3133872

1 INTRODUCTION

JavaScript is one of the most popular languages for writing web and mobile applications today. The language facilitates fast prototyping of ideas via dynamic typing. The runtime provides the means for fast iteration on those ideas via dynamic compilation. This fuels a fast edit-refresh cycle, which promises an immersive coding experience that is quite appealing to creative developers.

However, evolving and growing a JavaScript codebase is notoriously challenging. Developers spend a lot of time debugging silly mistakes—like mistyped property names, out-of-order arguments, references to missing values, checks that never fail due to implicit conversions, and so on—and worse, unraveling assumptions and guarantees in code written by others. In many other languages, this overhead is mitigated by having a layer of types over the code and building tools for the developer that use type information. For example, types can be used to identify common bugs and to document interfaces of libraries. Our aim is to bring such type-based tooling to JavaScript.

2475-1421/2017/10-ART48
https://doi.org/10.1145/3133872

1.1 Goals

In this paper, we present the design and implementation of Flow, a static type checker for JavaScript we have built and have been using at Facebook for the past three years.

The idea of using types to manage code evolution and growth in JavaScript (and related languages) is not new. In fact, several useful type systems have been built for JavaScript in recent years. The design and implementation of Flow are driven by the specific demands of real-world JavaScript development we have observed at Facebook and the industry at large.

- The type checker must be able to cover large parts of the codebase without requiring too many changes in the code. Developers want precise answers to code intelligence queries (the type of an expression, the definition reaching a reference, the set of possible completions at a point). Relatedly, they want to catch a large number of common bugs with few false positives.
- The type checker must provide very fast responses, even on a very large codebase. Developers do not want any noticeable "compile-time" latency in their normal workflow, because that would defeat the whole purpose of using JavaScript.

To meet these demands, we had to make careful choices and solve technical challenges in Flow that go beyond related existing systems.

- We precisely model common JavaScript idioms that appear pervasively in a modern JavaScript codebase. For example, Flow understands the pattern `x = x || 0` that, *e.g.*, initializes an optional parameter `x` in a function body. Handling a case like this necessitates support for type refinements: the system needs to recognize that the assigned value will not be **`null`** or **`undefined`**, *i.e.*, the type of the initial value of `x` must be *refined* to exclude *falsy* values.[1] (More examples are shown below.)
- At the same time, we *do not* focus on reflection and legacy patterns that appear in a relatively small fraction (that is also usually stable and well-tested). Today, tools like Babel convert modern JavaScript to (the more low-level) ES5 executed on browsers. Flow focuses on analyzing the source, instead of the target, of such translations (unlike many previous efforts that address ES5, or the even more low-level, and therefore harder, ES3).
- We modularize our constraint-based analysis and implement data structures, algorithms, and systems infrastructure for parallel computing, shared-memory communication, and incremental updates to scale to millions of lines of code while answering most queries in well under a second.

1.2 Overview

We now introduce the main ideas behind Flow's design and implementation. (A full description of Flow is not possible due to space constraints.)

Precise type checking. One of the main contributors of Flow's precision is path-sensitivity: the way types interact with runtime tests. The essence of many JavaScript idioms is to put together ad hoc sets of runtime values and to take them apart with shallow, structural (in)equality checks. In Flow, the set of runtime values that a variable may contain is described by its type, and a runtime test on that variable *refines* the type to a smaller set. This ability turns out to be quite powerful and general in practice.

In this paper, we formalize refinements in a core subset of JavaScript. The system is particularly interesting because of *the combination of mutable local variables and closures that capture them by reference*. Next, we illustrate this system via a series of examples (Figure 1).

[1] The values **`false`**, `0`, `""`, **`null`**, **`undefined`**, and **`NaN`** are *falsy*. All other values are *truthy*.

```
1  function pipe(x, f) { f(x); }
2  var hello = (s) => console.log("hello", s);
3  pipe("world", hello);
4  pipe("hello", null); // error

5  function pipe(x, f) {
6    if (f != null) { f(x); } // ok
7  }

8  var nil = { kind: "nil" };
9  var cons = (head, tail) => {
10   return { kind: "cons", head, tail };
11 }

12 function sum(list) {
13   if (list.kind === "cons") {
14     return list.head + sum(list.tail); // ok
15   }
16   return 0;
17 }
18 sum(cons(6, cons(7, nil)));

19 function merge(x) {
20   x = x || nil;
21   return x.kind; // ok
22 }

23 function havoc(x) {
24   function reset() { x = null; }
25   x = x || nil;
26   reset();
27   return x.kind; // error
28 }
```

Fig. 1. Modern JavaScript Examples

Higher-order functions (lines 1–4) are quite common in JavaScript. Unfortunately, it is also common to use `null` as a default for everything (line 4). In particular, this causes the dreaded "*null is not a function*" error to hit often. Fortunately, Flow finds these errors by following flows of `null` to calls in the code.

Checking for nullability is the idiomatic way to prevent such errors at runtime. (In JavaScript, the check `f != null` is equivalent to `f !== null && f !== undefined`, which additionally rules out `undefined`, commonly used to denote missing values.) Thankfully Flow understands that this code is safe. It refines the type of `f` to filter out `null` in line 6, and thus knows that `null` cannot reach the call. Many other idiomatic variants also work, such as `f && f(x)`, where `f` is checked to be truthy (ruling out `null`, `undefined` and other falsy values) before calling.

Refinements also power a common technique to encode algebraic data types in JavaScript, which are used quite widely (to manage actions and dispatchers, data and queries, *etc.* in user interface libraries). Records of different shapes have a common property that specifies the "constructor," and other properties dependent on the constructor value. These records are then analyzed by "pattern matching"—inspecting and branching on the constructor value.

For example, consider the encoding of lists in lines 8–11. A `sum` function (line 12) checks whether a list is non-empty before accessing properties specific to non-empty lists. Following the calls to `sum`, Flow knows that the parameter `list` in line 12 can contain both kinds of objects—those whose `kind` property is `"cons"`, and those for which it is `"nil"`. The latter ones are filtered out by refining the type of `list` in line 13, so that the only objects reaching the property accesses of `head` and `tail` in line 14 are guaranteed to have those properties. Thus, Flow knows that this code is safe. Without refinements, on the other hand, the analysis would have over-conservatively concluded that `nil` can also flow to the property accesses, leading to spurious type errors.

Refinements are tracked by a flow-sensitive analysis, and interact in interesting ways with variable assignment. The common idiom in line 20 of `merge` ensures that a variable has a non-`null` default. Flow models the assignment by merging the refined type of `x` with the type of `nil` and updating the type of `x` with it.

On the other hand, refinements can be *invalidated* by assignments, which can even happen indirectly via calls (line 26). While invalidating refinements is necessary for soundness, they should

be preserved as much as possible to avoid spurious type errors. Flow tracks variable assignments as *effects* for precise invalidation.

Refinements and their invalidation carry over to higher-order functions. We also have limited support for refining mutable object properties, but those refinements are invalidated aggressively (*i.e.*, our analysis is not heap-sensitive).

Behind the scenes, Flow relies on set-based analysis as a common low-level "assembly language" for encoding a wide variety of high-level analyses. Compared with pure unification, this affords far more precision, but is much less efficient (quasi-cubic vs. quasi-linear in program size). How do we scale such an analysis?

Fast type checking. The key to Flow's speed is modularity: the ability to break the analysis into file-sized chunks that can be assembled later. Fortunately, JavaScript is already written using files as modules, so we modularize our analysis simply by asking that modules have explicitly typed signatures. (We still infer types for the vast majority of code "local" to modules.) Coincidentally, developers consider this good software engineering practice anyway.

With modularity, we can aggressively parallelize our analysis. Furthermore, when files change, we can incrementally re-analyze only those files that depend on the changed files, and avoid re-analysis when their typed signatures have not changed. Together, these choices have helped scale the analysis to millions of lines of code.

Under the hood, Flow relies on a high-throughput low-latency systems infrastructure that enables distribution of tasks among parallel workers, and communication of results in parallel via shared memory. Combined with an architecture where the analysis of a codebase is updated automatically in the background on file system changes, Flow delivers near-instantaneous feedback as the developer edits and rebases code, even in a large repository.

1.3 Contributions

Overall, this paper (and our work on Flow) shows how, through careful design and implementation, type checking for JavaScript can be both extremely fast and precise enough in practice. We make the following contributions.

(1) We identify a lightweight form of type refinement as a crucial feature for supporting a variety of common JavaScript idioms in practice. We formalize inference to support refinements in a core fragment of JavaScript containing higher-order functions, mutable variables, runtime tests, and capture-by-reference. Knowing when to invalidate a refinement can be quite tricky in this setting (Sections 3.1 and 3.3). To the best of our knowledge, no prior work has formalized this subset of features in a type inference for JavaScript. We also discuss our implementation of type inference as a system of set-based constraints combined with unification for optimization (Section 3.6). We define a runtime semantics (Section 4) and prove our system safe (Section 5) with respect to it.
(2) We show how our inference system can be extended to check type annotations. Union types introduce some interesting complications (Section 6). We then identify a lightweight way to use type annotations to modularize our analysis, piggybacking on existing best practices in JavaScript development such as breaking a codebase into several small modules and documenting types at their boundaries (Section 7).
(3) We show how we can exploit modularity and dependency management to make Flow responsive at scale. We describe an algorithm for parallelizing and incrementally updating the analysis when files change (Section 8). We describe how we extend an existing system infrastructure for parallel computation and communication, and implement our algorithms on it to achieve high throughput and low latency (Section 9).

Expressions	e	::=	$x \mid n \mid x = e \mid (x) \Rightarrow \{s; \texttt{return}\ e\} \mid e_1(e_2)$
		$\mid$	$\{\texttt{f}_1 : e_1, \ldots, \texttt{f}_n : e_n\} \mid e.\texttt{f} \mid e_1.\texttt{f} = e_2$
		$\mid$	$p(x) \mid e_1 \,\&\&\, e_2 \mid e_1 \,||\, e_2 \mid \,!\, e$
Statements	s	::=	$e \mid \texttt{var}\ x = e \mid \texttt{if}\,(e)\,\{s_1\}\,\texttt{else}\,\{s_2\} \mid s_1; s_2 \mid \texttt{skip}$

Fig. 2. FlowCore Syntax

(4) We perform a thorough experimental evaluation of Flow on a codebase with millions of lines of JavaScript. Through key metrics we demonstrate the behavior of various stages of type checking at scale, and validate our hypotheses on the precision gained by refinement tracking and our low annotation footprint. (Section 10).

We conclude with a discussion of related work (Section 11) and of limitations and threats to validity (Section 12).

2 LANGUAGE FLOWCORE

We consider a minimal subset of JavaScript that includes functions, mutable variables, primitive values and records. Notably, we leave out data structures like dictionaries and arrays, as well as object-oriented features like **`this`**, methods, classes, and inheritance. These parts of the language are mostly orthogonal to understanding refinements. Their type inference, while interesting, is built on the same foundations, and behave more or less similarly to previous work—we can safely extend our model to include them, without significantly complicating our guarantees. Our focus is on formalizing type inference and refinement strengthening, with the exception of refinements on mutable fields that are not tracked through the heap. While compact, this fragment is expressive enough to model the examples of Section 1—which illustrate how Flow uses predicate refinements to reduce false positives, while remain sound with respect to variable updates.

2.1 Syntax

Figure 2 describes the language of *expressions* e and *statements* s. Here, n represents constants, and x and y range over program variables.

Expressions. We elide primitive operations (which may include arithmetic operations). Constants include, *e.g.*, numbers, strings, and **`undefined`**. The syntax $p(x)$ draws from a fixed, possibly infinite set of unary *predicates* p on x. These model dynamic checks, such as **`typeof`** `x === "number"`, `x === undefined`, `x` (testing if an expression is *truthy*), or model tests like `x.f === "nil"` on records. Note that in this system the last check does not imply a predicate on the value of `x.f`, but rather on `x` itself. The former would be a heap refinement, which Flow only supports in a limited fashion, and which is excluded from the formalism.

General-purpose functions (using the keyword `function`) are complicated in JavaScript: they can be additionally used as methods and as constructors. To simplify our exposition, we restrict our attention to *arrow* functions (essentially lambdas). We assume that a function body consists of a statement followed by the return of an expression. Functions that do not explicitly return anything can be thought of as implicitly returning **`undefined`**. (Flow's treatment of abnormal control flows via **`return`** is also interesting, but we omit it here.) We also include the logical conjunction (`&&`), disjunction (`||`) and negation (`!`) operators, as they are pervasive in JavaScript and inform our refinement strategy.

Type Vars	$\alpha, \beta, \gamma, \delta$	$\in$	$\mathcal{V}$
Type Literals	$\hat{\tau}$	$::=$	$b \mid \alpha \xrightarrow{\varepsilon} \tau \mid \{\mathtt{f}_1 : \alpha_1, \ldots, \mathtt{f}_n : \alpha_n\}$
Types	τ	$::=$	$\hat{\tau} \mid \tau_1 \sqcup \tau_2 \mid \alpha$
Effect Vars	ω	$\in$	$\mathcal{E}$
Effect Literals	$\hat{\varepsilon}$	$::=$	$\bot \mid x$
Effects	ε	$::=$	$\hat{\varepsilon} \mid \varepsilon_1 \sqcup \varepsilon_2 \mid \omega$
Type Uses	u_τ	$::=$	$\alpha \mid \mathrm{Call}(\tau \xrightarrow{\omega} \alpha) \mid \mathrm{Get}(\{\mathtt{f} : \alpha\}) \mid \mathrm{Set}(\{\mathtt{f} : \tau\}) \mid \mathrm{Pred}(P, \alpha)$
Effect Uses	u_ε	$::=$	$\omega \mid \mathrm{Havoc}(\Gamma)$
Predicates	P	$::=$	$p \mid \neg p$
Constraints	c	$::=$	$\tau \leq u_\tau \mid \varepsilon \leq u_\varepsilon$

Fig. 3. FlowCore Type, Effect and Constraint Syntax

Statements. We use **`var`** to introduce variables, and include statements for conditional execution and sequencing. We omit **`const`** because it is much simpler than **`var`**, since refinements never need to be invalidated. We also omit **`let`**-bound variables. Their main difference with **`var`**-bound variables is in scoping rules, so handling them does not add any insight to our type system overview. Finally, we omit **`while`**; although it can be encoded with **`if`** and recursion, Flow's treatment of it is more precise.

Assumption. We assume an α-renaming pre-pass over the program's AST that guarantees that each variable definition point (which is either a **`var`** statement or an arrow definition) introduces a unique variable identifier. This is a fairly straightforward transformation for any preprocessor that helps avoid unintentional capture of variables in exported closures.

2.2 Types, Effects and Constraints

The basic ingredients of our constraint system are *types* τ and *effects* ε. Their syntax is described in Figure 3.

Types. Types are ranged over by variables α, β, *etc.* taken from an enumerable set $\mathcal{V}$. The building blocks for constructing complex type structures are *type literals* $\hat{\tau}$. These include primitive types b (*e.g.*, `number`, `string`, and void for **`undefined`**), arrow types $\alpha \xrightarrow{\varepsilon} \tau$ for functions, and record types $\{\mathtt{f}_1 : \alpha_1, \ldots, \mathtt{f}_n : \alpha_n\}$. Arrow types are annotated with an effect ε which describes a set of names x that may be assigned in the function's body or transitively in code that is executed when calling this function. A more proper introduction of effects follows. Types also feature a binary operator $\sqcup$ denoting the union of types.

Effects. The effect we are interested in tracking here is variable updates. Each language term is associated with an effect, as we will see later in constraint generation. This is (roughly) the set of variables that are (re)assigned within this term. The base constructors of effects are the empty effect $\bot$ and variable symbols x, corresponding to the variables that are updated. Like types, effects also feature a binary operator $\sqcup$ denoting union of effects. Finally, effects are ranged over by variables ω taken from an enumerable set $\mathcal{E}$.

Environments. An *environment* Γ binds variables x to entries τ^{α}, meaning that its most recent assignment was of type τ, whereas the type variable α is used as the collective summary for all its (past, current, and future) assignments. Here τ is flow-sensitive—its value may change from one (flow-sensitive) environment to another—whereas α is invariant. We also distinguish between environment *extension*—denoted $\Gamma, x : \tau^{\alpha}$ (variable x is not bound in the original environment Γ), and environment *update*—denoted $\Gamma[\, x \mapsto \tau^{\alpha}\,]$ (variable x was bound in Γ).

Predicates. Key to our type refining process is the notion of predicates. A predicate P is a clause denoting a property of its implied argument. In our setting, syntactically it can be a *base* predicate p or its negation. Base predicates describe properties of constructed or primitive types. For the remaining sections we will keep these predicates abstract, but examples of these predicates are the ones implied by checks of the form `typeof ★ === "string"`, `typeof ★ === "number"`, `★.f === "null"`, *etc.*, where ★ is to be filled in with a program variable.

Constraints. A *constraint* c is a "flow" from a type τ (resp. effect ε) to a type *use* u_{τ} (resp. an effect *use* u_{ε}). Flows from types to type uses generalize the notion of subtyping. However, we chose to enforce some structural restrictions to the forms that can appear on the right-hand side of constraints, namely the uses. Type and effect variables can appear as uses themselves. We do not allow general types and effects to appear as uses. Instead they are wrapped by constructors that contain information about the operations that caused the generation of such constraints. Uses account for data flow through function calls (Call), object operations (Get, Set), control flow refinement (Pred), and refinement invalidation (Havoc).

The use $\mathsf{Call}(\tau \xrightarrow{\omega} \alpha)$ corresponds to a function call with argument type τ, resulting in type α; the effect variable ω models the effect of the target function. A constraint $\tau' \leq \mathsf{Call}(\tau \xrightarrow{\omega} \alpha)$ looks up the parameter, return, and effect of τ' and propagates τ to the parameter, the return to α, and the effect to ω. The uses for reading and writing to a field, $\mathsf{Get}(\{\mathtt{f} : \alpha\})$ and $\mathsf{Set}(\{\mathtt{f} : \tau\})$, are straightforward. A constraint $\tau' \leq \mathsf{Get}(\{\mathtt{f} : \alpha\})$ (resp. $\tau' \leq \mathsf{Set}(\{\mathtt{f} : \tau\})$) looks up the field $\mathtt{f}$ of τ' and propagates the result to α (resp. τ to the result). The use $\mathsf{Pred}(P,\ \alpha)$ is used to *refine* an incoming type using *predicate* P, resulting in fresh type variable α. In other words, a constraint $\tau \leq \mathsf{Pred}(P,\ \alpha)$ will only allow the parts of τ that satisfy P to flow to α. Finally, for refinement invalidation we introduce Havoc, which takes an environment argument Γ. A constraint $x \leq \mathsf{Havoc}\ (\Gamma)$ says that the variable x may be updated, so that any refinement involving x in Γ must be invalidated. This will be discussed later on in greater detail (Section 3.3).

3 CONSTRAINT SYSTEM

We present the static semantics of our formal fragment by means of a constraint generating type inference scheme. Our constraints encode type safety obligations that arise as values flow to operations throughout the program.

3.1 Constraint Generation

The core type inference judgments for expressions and statements in FlowCore are:

$$\Gamma \vdash e : \tau;\ \varepsilon;\ \psi \dashv \Gamma' \triangleright C \qquad\qquad \Gamma \vdash s : \varepsilon \dashv \Gamma' \triangleright C$$

The derivation of a judgment relies on a set of constraints C as proof obligations, which appear on the right of the $\triangleright$ symbol. We use $\cup$ for the union of two constraint sets and $\bigcup$ for the union of a number of constraint sets ranged over by the index in the subscript of the arguments set (*e.g.*, $\bigcup_{i=1}^{k} C_i$). For both expressions and statements this judgment is *flow-sensitive* which is achieved by introducing an output environment Γ', in addition to the input environment Γ. The set of variable

Expression Constraint Generation $\boxed{\Gamma \vdash e : \tau;\ \varepsilon;\ \psi \dashv \Gamma' \triangleright C}$

$$\frac{}{\Gamma \vdash n : b_n;\ \bot;\ \varnothing \dashv \Gamma \triangleright \varnothing}\ [\text{CG-Const}] \qquad \frac{\Gamma(x) = \tau^{\alpha}}{\Gamma \vdash x : \tau;\ \bot;\ x \mapsto \mathsf{truthy} \dashv \Gamma \triangleright \varnothing}\ [\text{CG-Var}]$$

$$\frac{\Gamma \vdash e : \tau;\ \varepsilon;\ \psi \dashv \Gamma' \triangleright C \qquad \Gamma'(x) = {\tau_0}^{\alpha}}{\Gamma \vdash x = e : \tau;\ \varepsilon \sqcup x;\ \psi \backslash x \dashv \Gamma'[x \mapsto \tau^{\alpha}] \triangleright C \cup \{\tau \leq \alpha\}}\ [\text{CG-Assign}]$$

$$\frac{\alpha, \overline{\alpha_i}\ \text{fresh} \qquad \Gamma_1 = \mathsf{erase}(\Gamma),\ x : \alpha^{\alpha},\ \overline{x_i : \mathsf{void}^{\alpha_i}} \qquad \overline{x_i} = \mathsf{locals}(s) \qquad \Gamma_1 \vdash \{s;\ \mathtt{return}\ e\} : \tau;\ \varepsilon \dashv \Gamma_2 \triangleright C}{\Gamma \vdash (x) \Rightarrow \{s;\ \mathtt{return}\ e\} : \alpha \xrightarrow{\varepsilon \backslash x, \overline{x_i}} \tau;\ \bot;\ \varnothing \dashv \Gamma \triangleright C}\ [\text{CG-Fun}]$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1;\ \varepsilon_1;\ \psi_1 \dashv \Gamma_1 \triangleright C_1 \\ \Gamma_1 \vdash e_2 : \tau_2;\ \varepsilon_2;\ \psi_2 \dashv \Gamma_2 \triangleright C_2 \qquad \alpha, \omega\ \text{fresh} \qquad \mathsf{widen}(\Gamma_2) = \Gamma_3 \triangleright C_3 \\ \varepsilon_1 \sqcup \varepsilon_2 \sqcup \omega = \varepsilon \qquad C_1 \cup C_2 \cup C_3 \cup \left\{\omega \leq \mathsf{Havoc}(\Gamma_3),\ \tau_1 \leq \mathsf{Call}(\tau_2 \xrightarrow{\omega} \alpha)\right\} = C\end{array}}{\Gamma \vdash e_1(e_2) : \alpha;\ \varepsilon;\ \varnothing \dashv \Gamma_3 \triangleright C}\ [\text{CG-Call}]$$

Fig. 4. Expression Constraint Generation in FlowCore (Variables and Functions)

names assigned in e or s is modeled by ε. The case of expressions has two additional byproducts: a *type* τ and a *predicate mapping* ψ. The latter includes bindings from names to predicates that must hold when e is truthy, and symbolic operations over them (explained later):

ψ	::=	$\varnothing$	empty mapping
	|	$x \mapsto P$	variable binding
	|	$\psi_1 \wedge \psi_2$	conjunction
	|	$\psi_1 \vee \psi_2$	disjunction
	|	$\neg\psi$	negation
	|	$\psi \backslash \varepsilon$	exclude effect

Below we describe constraint generation in more detail, starting from rules handling variables, functions, and calls (Figure 4).

Variables. The rules for reading and assigning a local variable (CG-Var and CG-Assign) involve looking up and updating the current type for the variable in the outgoing environment. This part is what makes this system *flow-sensitive*. A flow-insensitive system would use a single environment for each judgment. The assigned type would be merged to the same type used for the variable under update in the first place, making it less precise. In addition, reading a variable introduces a truthy predicate on it. This is useful under specific contexts such as when the variable is used as the condition part of an if-branch. Conversely, writing a variable forgets any refinement coming from expression e that concerns x.

Arrow Functions. Rule CG-Fun handles arrow functions by approximating the environment at the beginning with the *flow-insensitive erasure* of the current environment (since we do not know where this function will be called). The meta-function erase computes this new environment by mapping each $x : \tau^{\alpha}$ to $x : \alpha^{\alpha}$ (Figure 5). In addition, to capture the hoisting of variables defined within

Auxiliary Functions $\boxed{\mathsf{erase}(\Gamma) = \Gamma'}$ $\boxed{\mathsf{widen}(\Gamma) = \Gamma' \triangleright C}$

$$\frac{}{\mathsf{erase}(\cdot) = \cdot}\ [\text{T-Erase-E}] \qquad \frac{\mathsf{erase}(\Gamma) = \Gamma'}{\mathsf{erase}(\Gamma,\ x\colon \tau^{\alpha}) = \Gamma',\ x\colon \alpha^{\alpha}}\ [\text{T-Erase-C}]$$

$$\frac{}{\mathsf{widen}(\cdot) = \cdot \triangleright \varnothing}\ [\text{T-Widen-E}] \qquad \frac{\mathsf{widen}(\Gamma) = \Gamma_0 \triangleright C_0 \qquad \beta \text{ fresh}}{\mathsf{widen}(\Gamma,\ x\colon \tau^{\alpha}) = \Gamma_0,\ x\colon \beta^{\alpha} \triangleright C_0 \cup \{\, \tau \le \beta,\ \beta \le \alpha \,\}}\ [\text{T-Widen-C}]$$

Auxiliary Environment Operations $\boxed{\Gamma_1 \sqcup \Gamma_2 = \Gamma}$ $\boxed{\Gamma :: \psi = \Gamma' \triangleright C}$

$$\frac{}{\cdot \sqcup \cdot = \cdot}\ [\text{Env-Join-E}] \qquad \frac{\Gamma_1 \sqcup \Gamma_2 = \Gamma}{(\Gamma_1,\ x\colon {\tau_1}^{\alpha}) \sqcup (\Gamma_2,\ x\colon {\tau_2}^{\alpha}) = \Gamma,\ x\colon (\tau_1 \sqcup \tau_2)^{\alpha}}\ [\text{Env-Join-C}]$$

$$\frac{}{\Gamma :: \varnothing = \Gamma \triangleright \varnothing}\ [\text{Ref-E}] \qquad \frac{\Gamma(x) = \tau^{\alpha} \qquad \beta \text{ fresh}}{\Gamma :: x \mapsto P = \Gamma[\, x \mapsto \beta^{\alpha} \,] \triangleright \{\, \tau \le \mathsf{Pred}(P,\ \beta) \,\}}\ [\text{Ref-Single}]$$

$$\frac{\Gamma :: \psi_1 = \Gamma_1 \triangleright C_1 \qquad \Gamma_1 :: \psi_2 = \Gamma_2 \triangleright C_2}{\Gamma :: (\psi_1 \wedge \psi_2) = \Gamma_2 \triangleright C_1 \cup C_2}\ [\text{Ref-And}] \qquad \frac{\Gamma :: \psi_1 = \Gamma_1 \triangleright C_1 \qquad \Gamma :: \psi_2 = \Gamma_2 \triangleright C_2}{\Gamma :: (\psi_1 \vee \psi_2) = \Gamma_1 \sqcup \Gamma_2 \triangleright C_1 \cup C_2}\ [\text{Ref-Or}]$$

$$\frac{\Gamma :: \psi = \Gamma_1 \triangleright C_1 \qquad \mathsf{widen}(\Gamma_1) = \Gamma_2 \triangleright C_2 \qquad \Gamma_3 = \{\, x\colon \beta^{\tau} \mid x\colon \tau^{\alpha} \in \Gamma,\ x\colon \beta^{\alpha} \in \Gamma_2 \,\}}{\Gamma :: \psi \backslash \varepsilon = \Gamma_2 \triangleright C_1 \cup C_2 \cup \{\, \varepsilon \le \mathsf{Havoc}\,(\Gamma_3) \,\}}\ [\text{Ref-Effect}]$$

Fig. 5. Auxiliary Functions for Function Logistics and Environment Operations in FlowCore

the scope of the function to the beginning of the function body, we introduce the meta-function locals that takes as argument a statement s and returns all variable identifiers $\overline{x_i}$ declared in s. Each variable x_i is bound to the undefined type void (and a fresh general type α_i), since its definition is hoisted to the top of the function body and initialized to `undefined`. The inferred arrow type carries the effect ε of the body of the function. Note that we are removing the formal parameter x and local variables $\overline{x_i}$ from the effect as they are only visible within the body of the defined arrow.

Calls. Rule CG-Call handles calls. We approximate the outgoing environment with a *flow-sensitive widening* of the current environment (instead of pessimistically erasing everything in scope). The meta-function widen (Figure 5) computes this new environment Γ' by mapping each $x\colon \tau^{\alpha}$ to $x\colon \beta^{\alpha}$ where β is a fresh type variable such that $\tau \le \beta \le \alpha$. For any variable x that gets assigned during the function call, we must fall back to its erasure, *i.e.*, we must flow α to β. For now this is achieved by flowing the effect ω of the call to Havoc (Γ'). The actual erasure happens later at constraint propagation (Section 3.3), when the type of the receiver function is known and the incoming effect is no longer abstract. As we show in Section 3.3, when a function type flows to $\mathsf{Call}(\tau_2 \xrightarrow{\omega} \alpha)$, the effect ω is instantiated with the actual effect variables x carried over by the incoming function type. These variables trigger the erasure.

Environment Operations. Before delving into the remaining typing rules, we introduce some operations on environments (Figure 5).

Ther first one is *environment join* ($\sqcup$), a commutative operator that computes the *least upper bound* of a pair of environments with the same domain. Type entries bound to the same symbol in

Expression Constraint Generation $\boxed{\Gamma \vdash e : \tau;\ \varepsilon;\ \psi \dashv \Gamma' \triangleright C}$

$$\frac{\Gamma \vdash e_1 : \tau_1;\ \varepsilon_1;\ \psi_1 \dashv \Gamma_1 \triangleright C_1 \qquad \Gamma_1 :: \psi_1 = \Gamma_1' \triangleright C_2 \qquad \Gamma_1' \vdash e_2 : \tau_2;\ \varepsilon_2;\ \psi_2 \dashv \Gamma_2 \triangleright C_3 \quad \alpha_1 \text{ fresh} \qquad (\psi_1 \backslash \varepsilon_2) \wedge \psi_2 = \psi \qquad \Gamma_1 :: \neg\psi_1 = \Gamma_1'' \triangleright C_4 \qquad \Gamma_1'' \sqcup \Gamma_2 = \Gamma'}{\Gamma \vdash e_1 \,\&\&\, e_2 : \alpha_1 \sqcup \tau_2;\ \varepsilon_1 \sqcup \varepsilon_2;\ \psi \dashv \Gamma' \triangleright \bigcup_{i=1}^{4} C_i \cup \{\, \tau_1 \leq \mathsf{Pred}(\mathsf{falsy}, \alpha_1) \,\}} \text{ [CG-And]}$$

$$\frac{\Gamma \vdash e_1 : \tau_1;\ \varepsilon_1;\ \psi_1 \dashv \Gamma_1 \triangleright C_1 \qquad \Gamma_1 :: \neg\psi_1 = \Gamma_1' \triangleright C_2 \qquad \Gamma_1' \vdash e_2 : \tau_2;\ \varepsilon_2;\ \psi_2 \dashv \Gamma_2 \triangleright C_3 \quad \alpha_1 \text{ fresh} \qquad (\psi_1 \backslash \varepsilon_2) \vee \psi_2 = \psi \qquad \Gamma_1 :: \psi_1 = \Gamma_1'' \triangleright C_4 \qquad \Gamma_1'' \sqcup \Gamma_2 = \Gamma'}{\Gamma \vdash e_1 \,||\, e_2 : \alpha_1 \sqcup \tau_2;\ \varepsilon_1 \sqcup \varepsilon_2;\ \psi \dashv \Gamma' \triangleright \bigcup_{i=1}^{4} C_i \cup \{\, \tau_1 \leq \mathsf{Pred}(\mathsf{truthy}, \alpha_1) \,\}} \text{ [CG-Or]}$$

$$\frac{\Gamma \vdash e : \tau;\ \varepsilon;\ \psi \dashv \Gamma' \triangleright C}{\Gamma \vdash\, !\, e : \mathsf{bool};\ \varepsilon;\ \neg\psi \dashv \Gamma' \triangleright C} \text{ [CG-Not]} \qquad \frac{}{\Gamma \vdash p(x) : \mathsf{bool};\ \bot;\ x \mapsto p \dashv \Gamma \triangleright \varnothing} \text{ [CG-Pred]}$$

Fig. 6. Expression Constraint Generation in FlowCore (Logical Operations)

the input environments need to refer to the same program variable. This requirement allows us to assume that the general type of a variable x bound in both environments will be the same.

The next operation we define is *environment refinement* (::). The semantics of a refinement ψ is defined by how it refines environments through the constraint-producing judgment $\Gamma :: \psi = \Gamma' \triangleright C$, where an environment Γ is strengthened by the predicates in ψ and result in an environment Γ', potentially including fresh variables that are constrained in C. When ψ is $x \mapsto P$, we update the relevant binding in the environment Γ to a fresh type β that is the result of the predicate refinement of the initial type τ with P (Rule Ref-Single). The rules that handle the typical logical operators (Ref-And and Ref-Or) are straightforward.

Refinements can be invalidated by effects. In Rule Ref-Effect, we first refine Γ by ψ, and then apply the effect ε through the "havoc" mechanism on the resulting environment Γ_1. There is a slight discrepancy in the way this mechanism is applied in this case compared to function calls, since we only want to revert the effect of the refinement caused by ψ, and not fall back to the most general type. If "havoc" is triggered, then for every variable x bound in Γ_3, that happens to reach effect ε, we only flow type τ (that x was bound to in Γ before the refinement) to β, instead of the most general type α. It appears here that we are locally breaking our invariant on the form of environments, by allowing entries with types τ in the place of the most general type summary (exponent). This is a benign violation of our restriction on environments since the constructed environment Γ_3 is not used as the input environment in a type inference judgment, but rather as the argument to the "havoc" use. As we will see later, this context does not produce any flows towards τ. The use of an environment here is in fact a mere syntactic convenience.

Finally, we can have refinements with logical connectives. The negation of $x \mapsto p$ is simply $x \mapsto \neg p$. Otherwise, we push negations inward as much as possible, by applying the laws:

$$\neg(\psi_1 \wedge \psi_2) = \neg\psi_1 \vee \neg\psi_2 \qquad \neg(\psi \backslash \varepsilon) = \neg\psi \backslash \varepsilon$$
$$\neg(\psi_1 \vee \psi_2) = \neg\psi_1 \wedge \neg\psi_2 \qquad \neg(\neg\psi) = \psi$$

Logical operations. The rules of Figure 6 are interesting for their effect on predicate refinement. In Rule CG-And, e_2 is analyzed under the refinement ψ_1 (since otherwise it would not be evaluated). The type inferred for the entire expression contains components from both e_1 and e_2. From the former it contains type α_1 that is a version of τ_1 refined by the falsy predicate, since it corresponds to the case where e_1 is actually falsy. From the latter it includes the type τ_2 as is. For the output environment

Expression Constraint Generation $\boxed{\Gamma \vdash e : \tau ; \varepsilon ; \psi \dashv \Gamma' \triangleright C}$

$$\frac{\Gamma \equiv \Gamma_0 \qquad \forall i \in [1,n] \,.\, \Gamma_{i-1} \vdash e_i : \tau_i ; \varepsilon_i ; \psi_i \dashv \Gamma_i \triangleright C_i \qquad \alpha_i \text{ fresh}}{\Gamma \vdash \{\overline{\mathsf{f}_i : e_i}\} : \left\{\overline{\mathsf{f}_i : \alpha_i}\right\} ; \bigsqcup_{i=1}^{n} \varepsilon_i ; \varnothing \dashv \Gamma_n \triangleright \bigcup_{i=1}^{n} C_i \cup \bigcup_{i=1}^{n} \{ \tau_i \leq \alpha_i \}} \text{ [CG-Rec]}$$

$$\frac{\Gamma \vdash e : \tau ; \varepsilon ; \psi \dashv \Gamma' \triangleright C \qquad \alpha \text{ fresh}}{\Gamma \vdash e.\mathsf{f} : \alpha ; \varepsilon ; \varnothing \dashv \Gamma' \triangleright C \cup \{ \tau \leq \mathsf{Get}\,(\{\mathsf{f} : \alpha\}) \}} \text{ [CG-FdRd]}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 ; \varepsilon_1 ; \psi_1 \dashv \Gamma_1 \triangleright C_1 \qquad \Gamma_1 \vdash e_2 : \tau_2 ; \varepsilon_2 ; \psi_2 \dashv \Gamma_2 \triangleright C_2}{\Gamma \vdash e_1.\mathsf{f} = e_2 : \tau_2 ; \varepsilon_1 \sqcup \varepsilon_2 ; \varnothing \dashv \Gamma_2 \triangleright C_1 \cup C_2 \cup \{ \tau_1 \leq \mathsf{Set}\,(\{\mathsf{f} : \tau_2\}) \}} \text{ [CG-FdWr]}$$

Fig. 7. Expression Constraint Generation in FlowCore (Records)

Statement Constraint Generation $\boxed{\Gamma \vdash s : \varepsilon \dashv \Gamma' \triangleright C}$

$$\frac{\Gamma \vdash e : \tau ; \varepsilon ; \psi \dashv \Gamma' \triangleright C}{\Gamma \vdash e : \varepsilon \dashv \Gamma' \triangleright C} \text{ [CG-Exp]} \qquad \frac{\Gamma \vdash x = e : \tau ; \varepsilon ; \psi \dashv \Gamma' \triangleright C}{\Gamma \vdash \mathsf{var}\ x = e : \varepsilon \dashv \Gamma' \triangleright C} \text{ [CG-VarDecl]}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau ; \varepsilon ; \psi \dashv \Gamma' \triangleright C_1 \qquad \Gamma' :: \psi = \Gamma_1 \triangleright C_2 \\ \Gamma_1 \vdash s_1 : \varepsilon_1 \dashv \Gamma_1' \triangleright C_3 \qquad \Gamma' :: \neg\psi = \Gamma_2 \triangleright C_4 \qquad \Gamma_2 \vdash s_2 : \varepsilon_2 \dashv \Gamma_2' \triangleright C_5 \qquad \Gamma_1' \sqcup \Gamma_2' = \Gamma''\end{array}}{\Gamma \vdash \mathtt{if}\,(e)\,\{s_1\}\,\mathtt{else}\,\{s_2\} : \varepsilon \sqcup \varepsilon_1 \sqcup \varepsilon_2 \dashv \Gamma'' \triangleright \bigcup_{i=1}^{5} C_i} \text{ [CG-If]}$$

$$\frac{\Gamma \vdash s_1 : \varepsilon_1 \dashv \Gamma_1 \triangleright C_1 \qquad \Gamma_1 \vdash s_2 : \varepsilon_2 \dashv \Gamma_2 \triangleright C_2}{\Gamma \vdash s_1 ; s_2 : \varepsilon_1 \sqcup \varepsilon_2 \dashv \Gamma_2 \triangleright C_1 \cup C_2} \text{ [CG-Seq]}$$

Fig. 8. Statement Constraint Generation in FlowCore

we follow a similar strategy. The component that corresponds to e_1's output environment will be refined with $\neg\psi_1$, since otherwise we would be using the environment corresponding to e_2. With respect to the output predicate mapping, parts of ψ_1 that apply on names written in e_2 are forgotten when taking the conjunction with ψ_2. Rule CG-Or is the dual of the above rule, and works similarly. Finally, rules CG-Not and CG-Pred are straightforward. The former just negates the refinement and the latter introduces a refinement from a runtime test p.

Records. The rules of Figure 7 for record type inference are mostly routine. During record creation the initializer types flow to the newly constructed record literal type. Subsequent assignments of type τ to a field f widen the type of f by introducing flows to the use $\mathsf{Set}\,(\{\mathsf{f} : \tau\})$.

In practice, Flow follows a slightly stricter approach. It "fixes" the type of an object at initialization and checks that all subsequent writes adhere to this type. This essentially amounts to checking for type annotations which is out of scope in this section of type inference.

Statements. The main difference compared to the respective expression rule is the omission of the assigned type and the refinement predicate. Rule CG-VarDecl reuses the rule for assignment that we saw earlier, since due to variable hoisting, x is already in scope. Rule CG-If handles conditional statements. This rule uses the refinement ψ for the conditional expression e to refine

the environments that are used to check each branch, with the appropriate sign in each case. The output environment is the join of the environments at the end of each branch.

3.2 Example

We now examine how the rules of Figures 4 – 8 handle the code in lines 8 – 28 in Figure 1. In the following we keep the produced type bindings on the left and constraint sets on the right. Whenever, a general type (exponent) is not made explicit, this means that it's not important for that particular binding. Also, to avoid clutter, we do not define a new environment for each program point, but rather introduce different versions for variables that get updated or refined.

By applying Rule CG-Rec on line 8:

$$\texttt{nil}\colon \{\texttt{kind} : \alpha_1\} \qquad C \supseteq \{\,\texttt{"nil"} \le \alpha_1\,\} \tag{1}$$

Here, `"nil"` is the string literal type denoting the exact string `"nil"`. For the function `cons` (lines 9 – 11) we get

$$\texttt{cons}\colon (\alpha_2,\ \alpha_3) \to O \qquad C \supseteq \{\,\texttt{"cons"} \le \alpha_4,\ \alpha_2 \le \alpha_5,\ \alpha_3 \le \alpha_6\,\} \tag{2}$$

where $O \doteq \{\texttt{kind} : \alpha_4,\ \texttt{head} : \alpha_5,\ \texttt{tail} : \alpha_6\}$. We also define $\tau_{\texttt{cons}} \doteq (\alpha_2,\ \alpha_3) \to O$. The function's effect is empty, so omitted here. Moving on to function `sum`, before checking its body we introduce bindings for the (recursive) function itself and its parameter:

$$\texttt{sum}\colon \alpha_7 \to \tau_r,\ \texttt{list}\colon \alpha_7 \tag{3}$$

We define $\tau_{\texttt{sum}} \doteq \alpha_7 \to \tau_r$. Checking the conditional in line 13, `list` gets a more precise type, and is referred to as $\texttt{line}_1$ inside the then-branch:

$$\texttt{list}_1\colon \beta_7 \qquad C \supseteq \{\,\alpha_7 \le \mathsf{Pred}(p_c,\ \beta_7)\,\} \tag{4}$$

Here, $p_c \doteq \star.\texttt{kind}\ \texttt{===}\ \texttt{"cons"}$ is the predicate of exact equality of the field `kind` with the string `"cons"`. The uses of `list` in line 14 produce the following constraints (here we focus on the interesting uses *i.e.*, the two field accesses and the call):

$$C \supseteq \left\{ \begin{array}{l} \beta_7 \le \mathsf{Get}\,(\{\texttt{head} : \gamma_1\}), \\ \beta_7 \le \mathsf{Get}\,(\{\texttt{tail} : \gamma_2\}), \\ \tau_{\texttt{sum}} \le \mathsf{Call}(\gamma_2 \to \delta_1) \end{array} \right\} \tag{5}$$

We omit the constraints pertinent to the return statements, since they are not crucial in this example. The compound calls in line 18 further produce the constraints (starting from deeper nesting levels):

$$C \supseteq \{\,\tau_{\texttt{cons}} \le \mathsf{Call}((\texttt{number},\ \{\texttt{kind} : \alpha_1\}) \to \delta_2)\,\} \tag{6}$$

$$C \supseteq \{\,\tau_{\texttt{cons}} \le \mathsf{Call}((\texttt{number},\ \delta_2) \to \delta_3)\,\} \tag{7}$$

$$C \supseteq \{\,\tau_{\texttt{sum}} \le \mathsf{Call}(\delta_3 \to \delta_4)\,\} \tag{8}$$

In function `merge`, let $\texttt{x}_1$ correspond to the initial value for `x` and $\texttt{x}_2$ to the value after the update in line 20. Below, the first three constraints correspond to the use of the `||` operator and the last one to the field access in line 21:

$$\texttt{x}_1\colon {\alpha_8}^{\alpha_8},\ \texttt{x}_2\colon {\alpha_{11}}^{\alpha_8} \qquad C \supseteq \left\{ \begin{array}{l} \alpha_8 \le \mathsf{Pred}(\mathsf{truthy},\ \beta_8), \\ \beta_8 \sqcup \tau_{\texttt{nil}} \le \alpha_{11}, \\ \alpha_{11} \le \mathsf{Get}\,(\{\texttt{kind} : \alpha_{10}\}) \end{array} \right\} \tag{9}$$

$$\{\, \tau \le \alpha,\ \alpha \le u_\tau \,\} \subseteq C \implies \tau \le u_\tau \in C \qquad \text{(CP-Trans-T)}$$
$$\{\, \varepsilon \le \omega,\ \omega \le u_\varepsilon \,\} \subseteq C \implies \varepsilon \le u_\varepsilon \in C \qquad \text{(CP-Trans-E)}$$
$$\tau_1 \sqcup \tau_2 \le u_\tau \in C \implies \{\, \tau_1 \le u_\tau,\ \tau_2 \le u_\tau \,\} \subseteq C \qquad \text{(CP-Join-T)}$$
$$\varepsilon_1 \sqcup \varepsilon_2 \le u_\varepsilon \in C \implies \{\, \varepsilon_1 \le u_\varepsilon,\ \varepsilon_2 \le u_\varepsilon \,\} \subseteq C \qquad \text{(CP-Join-E)}$$
$$\alpha \xrightarrow{\varepsilon} \tau \le \mathsf{Call}(\tau' \xrightarrow{\omega} \beta) \in C \implies \{\, \tau' \le \alpha,\ \tau \le \beta,\ \varepsilon \le \omega \,\} \subseteq C \qquad \text{(CP-Call)}$$
$$x \le \mathsf{Havoc}\,(\Gamma,\ x\colon \beta^\alpha) \in C \implies \alpha \le \beta \in C \qquad \text{(CP-Havoc)}$$
$$\hat{\tau} \le \mathsf{Pred}(P,\ \alpha) \in C \wedge \mathsf{check}\,(\hat{\tau},\ P) \implies \hat{\tau} \le \alpha \in C \qquad \text{(CP-P-Base)}$$
$$\{\, \tau \le \alpha,\ \tau'\langle\alpha\rangle^+ \le \mathsf{Pred}(P,\ \beta) \,\} \subseteq C \implies \tau'\langle\tau\rangle^+ \le \mathsf{Pred}(P,\ \beta) \in C \qquad \text{(CP-P-Trans)}$$
$$\{\ldots, \mathsf{f} : \alpha, \ldots\} \le \mathsf{Get}\,(\{\mathsf{f} : \beta\}) \in C \implies \alpha \le \beta \in C \qquad \text{(CP-Get)}$$
$$\{\ldots, \mathsf{f} : \alpha, \ldots\} \le \mathsf{Set}\,(\{\mathsf{f} : \tau\}) \in C \implies \tau \le \alpha \in C \qquad \text{(CP-Set)}$$

Fig. 9. Constraint Propagation in FlowCore

Finally, function havoc in lines 23–28 is similar to `merge` (so we won't repeat the common parts), but additionally, defines a function `reset`, that assigns **`null`** to `x`. Crucially, the type of `x` inside `reset` has been erased to α_8:

$$\texttt{reset}\colon () \xrightarrow{\texttt{x}} \mathsf{void} \qquad\qquad C \supseteq \{\, \texttt{null} \le \alpha_8 \,\} \tag{10}$$

The call to `reset` in line 26 needs to handle the function's effect, so a fresh variable ω is generated:

$$C \supseteq \left\{ \begin{array}{l} () \xrightarrow{\texttt{x}} \mathsf{void} \le \mathsf{Call}(() \xrightarrow{\omega} \mathsf{void}), \\ \omega \le \mathsf{Havoc}\,(\Gamma[\,\texttt{x} \mapsto {\alpha_{11}}^{\alpha_8}\,]) \end{array} \right\} \tag{11}$$

For the moment, we have merely constructed a flow network, but haven't reached any critical conclusions. In the next section, we'll see how we can use these facts to discover inconsistencies, and what guarantees we get if we do not find any.

3.3 Propagation

Thinking of our system as a dataflow analysis framework, constraint generation amounts to setting up a flow network. The next step is to allow the system to stabilize under a set of appropriate flow functions. This latter part is called *constraint propagation* and corresponds to exploring *all* potential data-flow paths and finding inconsistencies in them. Decomposing complex constraints intro simpler ones is done by the rules shown in Figure 9. We say that a constraint set C is in *closed form*, if it is closed with respect to these rules. In practice, we keep our constraint sets in closed form at all times during constraint generation; that is, for every new constraint that gets generated, we apply all eligible propagation rules until we reach a fixpoint.

If we consider the elements of C as subtyping constraints, then these rules amount to subtyping rules. Rules CP-Trans-T and CP-Trans-E express transitivity for types and effect, respectively. CP-Join-T and CP-Join-E decompose as usual flows from joins of elements.

Rule CP-Call decomposes the flow of an arrow type to a calling context. Note that the incoming arrow type has a type variable α as the parameter type, since this is the form in which it is produced by CG-Fun. Also by the CG-Call the effect and return type portion of the calling use are also variables. Handling this flow propagates three new flows: (i) the argument's type τ' flows to the parameter type variable α, (ii) the return type τ flows to the call-site's type β, and (iii) the function's

effect ε flows to the call's effect variable ω. This last byproduct often triggers the "havoc" mechanism, which carries out the task of applying a function's effect on the variables that are updated by it.

Rule CP-Havoc recovers soundness by restoring the conservative types for variables that are updated through function calls (CG-Call) or are reassigned during conditional type refinement (Ref-Effect). Lets assume the havoc operation was introduced due to the former rule. (The latter case works similarly.) When CP-Havoc is triggered, our analysis has determined that variable x gets updated in the called function, and so entries bound to x in the environment after the function call should be conservatively approximated. Of course, this rule is only meaningful if x is bound in that environment. Otherwise this effect can be ignored. Effectively, this corresponds to erasing the type of the binding $x\colon \beta^{\alpha}$, by generating a flow from the flow-insensitive type α to β. This process may happen far away from the actual call-site, which exemplifies the global character of the type inference. An observant reader might notice that the environment argument of Havoc has entries of the form β^{α}. We can guarantee that this is the only possible form, by construction of the respective flows in rules CG-Call and Ref-Effect. In both cases this happens after a widening operation, which guarantees that the base of the environment entry is a type variable.

Rule CP-P-Base handles predicate refinement. The intuition here is that $\hat{\tau}$ should flow to α, if it succeeds in the check implied by P, *i.e.*, if check $(\hat{\tau},\ P)$ is true. We have kept the representation of base predicates abstract, and so we will do with the definition of check. In general, check should be able to decide if $\hat{\tau}$ satisfies P by inspecting its top-level constructor (for checks like `typeof ⋆ === "string"`) or one of its fields.

Rule CP-P-Trans is a technical one. It allows parts of types under refinement to be concretized. In $\tau'\langle\alpha\rangle$, the form $\tau'\langle\rangle$ is a *type context*, *i.e.*, a type with a "hole" that is filled in with α, for example $\{\mathtt{f} : \langle\rangle\}$. While rule CP-Trans-T will fail to instantiate α, CP-P-Trans allows type variables appearing under a type constructor (*e.g.*, the object constructor) to be instantiated. However, not all substitutions are allowed, but only the ones where α is in a positive position with respect to type *polarity* [Dolan and Mycroft 2017; Pottier 1998][2]. The reason we require type variable α to appear in a positive position is to abide by our restriction that type joins cannot appear at the right-hand side of constraints. If we allowed the replacement of α from τ in any part of τ', this could potentially break this invariant in a later propagation. We will also see the importance of this rule in the upcoming example.

Finally, Rules CP-Get and CP-Set handling record field access and update are standard.

3.4 Consistency

The goal of running constraint generation and propagation is to eventually discover inconsistencies in the saturated constraint set. These effectively correspond to potential bugs in the use of the various operators, for example they could correspond to the case of a non-function value reaching the receiver position of a call. Below we present a formal description of consistency.

Definition 3.1 (Consistency). A closed constraint set is *consistent* if it does not contain any constraints in one of the forms:

- $\hat{\tau} \le \mathsf{Call}(\tau \xrightarrow{\omega} \alpha)$ where $\hat{\tau}$ is *not* an arrow type (or an arrow-like type, *e.g.*, the type of a constructor object).
- $\hat{\tau} \le \mathsf{Set}\,(\{\mathtt{f} : \tau\})$ or $\hat{\tau} \le \mathsf{Get}\,(\{\mathtt{f} : \alpha\})$ where $\hat{\tau}$ is *not* an record type literal (or an object-like type) containing $\mathtt{f}$.

If our analysis finds an inconsistency, then this leads to an error report. Otherwise, if no inconsistency can be found then the input program enjoys the safety guarantees of Theorem 5.2.

[2]The extended version [Chaudhuri et al. 2017, Section A.4] includes a formal definition of polarity and type contexts.

3.5 Example

We continue where we left off in the example of Section 3.1, by applying the rules from Figure 9 on C, in order to discover inconsistencies or prove the absence thereof.

Use of predicates. We start by applying CP-Call on the calls of (5), (6), (7), (8), and the respective function definitions:

$$C \supseteq \{\, \gamma_2 \leq \alpha_7,\ \tau_r \leq \delta_1 \,\} \tag{12}$$
$$C \supseteq \{\, \texttt{number} \leq \alpha_2,\ \{\texttt{kind} : \alpha_1\} \leq \alpha_3,\ O \leq \delta_2 \,\} \tag{13}$$
$$C \supseteq \{\, \texttt{number} \leq \alpha_2,\ \delta_2 \leq \alpha_3,\ O \leq \delta_3 \,\} \tag{14}$$
$$C \supseteq \{\, \delta_3 \leq \alpha_7,\ \tau_r \leq \delta_4 \,\} \tag{15}$$

Now lets focus on the interesting case of handling the getters of (5). By transitivity (CP-Trans-T) using (14), (15) and (4), the record type O flows to the predicate use:

$$C \supseteq \{\, \{\texttt{kind} : \alpha_4,\ \texttt{head} : \alpha_5,\ \texttt{tail} : \alpha_6\} \leq \mathsf{Pred}(p_c,\ \beta_7) \,\} \tag{16}$$

We use Rule CP-P-Trans on (2) and (16) to obtain:

$$C \supseteq \{\, \{\texttt{kind} : \texttt{"cons"},\ \texttt{head} : \alpha_5,\ \texttt{tail} : \alpha_6\} \leq \mathsf{Pred}(p_c,\ \beta_7) \,\} \tag{17}$$

This is now a successful test since the string literal type `"cons"` of field `kind` satisfies p_c and so:

$$C \supseteq \{\, \{\texttt{kind} : \texttt{"cons"},\ \texttt{head} : \alpha_5,\ \texttt{tail} : \alpha_6\} \leq \beta_7 \,\} \tag{18}$$

Flow has thus discovered a path in which a "cons" object reaches the field accesses of line 14. However, this latest constraint has enabled new flows that could cause inconsistencies, *e.g.*, the recursive calls to `sum` on the `tail` of `list`. By (18) and (5), and applying CP-Trans-T and CP-Get:

$$C \supseteq \{\, \alpha_6 \leq \gamma_2 \,\} \tag{19}$$

Indeed, by combining (13), (2), (19), (12) and (4) with CP-Trans-T and the result with (1) with CP-P-Trans:

$$C \supseteq \{\, \{\texttt{kind} : \texttt{"nil"}\} \leq \mathsf{Pred}(p_c,\ \beta_7) \,\} \tag{20}$$

This test, however, will fail, as it would at runtime, and so the "nil" object will not reach the getter for `head` or `tail` through α_6. Without the predicate refinement filtering out "nil" objects, we would have introduced a false positive.

Refinements and Mutation. Last, we illustrate how Flow handles functions `merge` and `havoc`. We start by processing (11) with CP-Call and then CP-Havoc, which yields:

$$C \supseteq \{\, \alpha_8 \leq \alpha_{11} \,\} \tag{21}$$

This allows the **`null`** from the `reset` function to find its way to α_{11} from (10) and from there to the "get" operation through (9):

$$C \supseteq \{\, \texttt{null} \leq \mathsf{Get}\,(\{\texttt{kind} : \alpha_{10}\}) \,\} \tag{22}$$

This latter constraint signals a consistency violation, keeping Flow sound with respect to variable updates that invalidate prior refinements.

Runtime Expressions	e	::=	$\ldots \mid \ell$
Values	v	::=	$n \mid \ell$
Heap Values	$\dot{v}$	::=	$v \mid \langle \mathrm{L}, (x) \Rightarrow M \rangle \mid \{\mathtt{f}_1 : v_1, \ldots, \mathtt{f}_n : v_n\}$
Evaluation Contexts	E	::=	$\langle\,\rangle \mid x = \mathrm{E} \mid \mathrm{E}(e) \mid \ell(\mathrm{E}) \mid \mathrm{E}\,\&\&\,e \mid \mathrm{E} \mid\mid e \mid$
		$\mid$	$\{\mathtt{f}_1 : v_1, \ldots, \mathtt{f}_k : \mathrm{E}_k, \ldots, \mathtt{f}_n : e_n\}$
		$\mid$	$!\,\mathrm{E} \mid \mathrm{E}.\mathtt{f} \mid \mathrm{E}.\mathtt{f} = e \mid v.\mathtt{f} = \mathrm{E} \mid \mathtt{var}\ x = \mathrm{E}$
		$\mid$	$\mathtt{if}\,(\mathrm{E})\,\{s_1\}\,\mathtt{else}\,\{s_2\} \mid \mathtt{return}\ \mathrm{E} \mid \mathrm{E}; s$

Heaps	H	::=	$\cdot \mid \mathrm{H}, \ell \mapsto \dot{v}$
Stacks	X	::=	$\cdot \mid \mathrm{X}, \mathrm{L}.\mathrm{E}$
Stores	L	::=	$\cdot \mid \mathrm{L}, x \mapsto \ell$

States	S	::=	$\langle \mathrm{H}; \mathrm{X}; \mathrm{L} \rangle$
Configurations	R	::=	$S; e \mid S; s \mid S; M$

Fig. 10. Runtime Definitions in FlowCore

3.6 Implementation of Type Inference

A set of flow constraints C can be thought of as a constraint graph, where variables, literals and uses are the nodes and the constraints among them are the edges. In this section, we briefly discuss how we represent constraint graphs and compute their closure efficiently. Let us refer to type and effect variables as "unknowns." Following Pottier [2001], the constraint graph maps each unknown to a set of lower bounds and a set of upper bounds, each of which contains the unknown itself. The transitive propagation rules are specialized to exploit this structure to efficiently keep the constraint graph in closed form.

However, equality constraints are quite inefficient in this system: they are represented as a pair of subset constraints, which causes a cubic blowup in the transitive propagation rules. On the other hand, equality constraints are quite useful and common in Flow. They arise due to invariant typing of object properties, array elements, and type arguments of polymorphic classes. They directly model equations expressed by type aliasing. Finally, even though we formalize CP-Havoc with a constraint of the form $\alpha \leq \beta$, we can replace it without loss of generality with $\alpha = \beta$.

To address the inefficiency, we generalize the constraint graph by considering each unknown to be in an equivalence class containing other unknowns it is unified with, and mapping each equivalence class to either "unresolved" bounds (like Pottier [2001]), or to a "resolved" type or effect (as in unification). The transitive propagation rules generalize in a straightforward way. Overall, this simple optimization leads to $O(n)$ reduction in space and time complexity.

4 RUNTIME SEMANTICS

Before we describe our safety result (Section 5) we present the runtime semantics for the formal fragment of Section 2, which is heavily based on that used by Rastogi et al. [2015] that cover a subset of JavaScript, emphasizing on features of interest, while abstracting away non-crucial ones.

4.1 Definitions

Figure 10 contains the definitions for runtime configurations in FlowCore.

Runtime Values. To account for heap-allocated values, we introduce *locations* ℓ that index runtime heaps. Together with constants they synthesize runtime *values*, which are normal form as far as execution is concerned. Locations are also added to the set of expressions in our runtime language along with all other expression forms introduced earlier.

Runtime State. There are three constituent parts that compose a *runtime state* S. The first part is the *heap* H, which includes bindings from locations to *heap values* $\dot{v}$, which in turn are either values, closures, or heap objects. A *closure* is a pair containing a store L that binds all external variables available at the point of definition of the arrow function (capture by reference), and the function's code, which is a statement succeeded by a returned expression. The second part of the runtime state is the *stack* X, that contains a list of stack frames. Each stack frame includes a store containing the variables bound in the stack frame at the time execution left that frame, and an evaluation context E that holds the context that execution would jump into when returning to that stack frame. Evaluation contexts are defined in the usual way having the same structure as expressions or statements but with a hole $\langle\,\rangle$ at the position of the term that is about to be evaluated next. Finally, the runtime state includes a *store* L, that comprises bindings of variable names to locations to allow closures to capture values by reference.

Runtime Configurations. We write our *runtime configurations* S (*i.e.*, programs under execution) as pairs that contain a runtime state S, and a language term, which can either be an expression e, a statement s, or a function body $\{s;\ \texttt{return}\ e\}$. We conflate the notions of expressions and function bodies into a common notion using the symbol M, for compactness in stating our results.

4.2 Reduction Rules

Figure 11 contain a small-step operational semantics for programs in FlowCore. The rules can have the forms: $S;\ e \longrightarrow S';\ e'$ and $S;\ s \longrightarrow S';\ s'$.

Next we describe some of the most interesting rules. Rule RT-Var shows the indirection in dereferencing variables. First the store L is looked up and then the resulting location is used to access the heap H. Similarly variable assignments have to go through the same process in Rule RT-Asgn. Here, symbol $\triangleleft$ denotes the update of state S with the new heap H′.

When evaluating arrows, the current store S.L is saved as part of the created closure, along with the code M of the function (Rule RT-Arr). This store is restored when the function is called (Rule RT-Call). The new store L′ that will be used in the new stack frame also includes a binding for the function parameter x and bindings from all variables $\overline{x_i}$ defined in the body M, since their definition is hoisted to the top of the function scope. We use metavariable locals to extract these variables. All new variables are bound to fresh locations ℓ_i. Locals have not been initialized yet, so their locations are bound to **`undefined`** in the heap H′. Finally, a new stack frame L.E is pushed on the existing stack X as we enter the new function context. After returning from this function, execution will return to E (Rule RT-Ret). The rest of the expression reduction rules are routine.

5 METATHEORY

In order to prove type safety for our type system we first introduce a declarative type system that closely matches the intuition of the type inference system described in Section 3. Based on the declarative system we then formulate a type safety argument for the above language fragment via a progress and a preservation theorem [Wright and Felleisen 1994], that connect type checking with the runtime semantics of Section 4. Essentially, we establish the fact that if a program has been checked with the above algorithm and has been found consistent, then its execution will not lead to uncaught type errors (*e.g.*, "undefined is not a function"). Introducing this intermediate step in our metatheory is not mandatory, but it vastly reduces the complexity of reasoning about type safety, compared to the inference version of Section 3.

Expression and Statement Reduction Rules $\boxed{S;\, M \longrightarrow S';\, M'}$ $\boxed{S;\, s \longrightarrow S';\, s'}$

$$\frac{\langle \mathrm{H};\, \mathrm{X};\, \mathrm{L}\rangle;\, e \longrightarrow \langle \mathrm{H}';\, \mathrm{X};\, \mathrm{L}'\rangle;\, e'}{\langle \mathrm{H};\, \mathrm{X};\, \mathrm{L}\rangle;\, \mathrm{E}\langle e\rangle \longrightarrow \langle \mathrm{H}';\, \mathrm{X};\, \mathrm{L}'\rangle;\, \mathrm{E}\langle e'\rangle}\ [\text{RT-Ectx}] \qquad \frac{}{S;\, x \longrightarrow S;\, S.\mathrm{H}(S.\mathrm{L}(x))}\ [\text{RT-Var}]$$

$$\frac{\mathrm{H}' = \mathrm{H}[\,\mathrm{L}(x) \mapsto v\,]}{S;\, x = v \longrightarrow S \triangleleft \mathrm{H}';\, v}\ [\text{RT-Asgn}] \qquad \frac{\ell \text{ fresh} \qquad \mathrm{H}' = \mathrm{H},\, \ell \mapsto \langle S.\mathrm{L},\, (x) \Rightarrow M\rangle}{S;\, (x) \Rightarrow M \longrightarrow S \triangleleft \mathrm{H}';\, \ell}\ [\text{RT-Arr}]$$

$$\frac{\begin{array}{rll} \mathrm{H}(\ell) = \langle \mathrm{L}_0,\, (x) \Rightarrow M\rangle & \ell',\, \overline{\ell_i} \text{ fresh} & \overline{x_i} = \mathsf{locals}(M) \\ \mathrm{H}' = \mathrm{H},\, \ell' \mapsto v,\, \overline{\ell_i \mapsto \mathsf{undefined}} & \mathrm{X}' = \mathrm{X},\, \mathrm{L.E} & \mathrm{L}' = \mathrm{L}_0,\, x \mapsto \ell',\, \overline{x_i \mapsto \ell_i} \end{array}}{\langle \mathrm{H};\, \mathrm{X};\, \mathrm{L}\rangle;\, \mathrm{E}\langle \ell(v)\rangle \longrightarrow \langle \mathrm{H}';\, \mathrm{X}';\, \mathrm{L}'\rangle;\, M}\ [\text{RT-Call}]$$

$$\frac{S \equiv \langle \mathrm{H};\, \mathrm{X};\, \mathrm{L}\rangle \qquad \dot{v} = \mathrm{H}(\mathrm{L}(x))}{S;\, p(x) \longrightarrow S;\, \delta_p(\dot{v})}\ [\text{RT-Pred-Var}] \qquad \frac{\mathsf{truthy}(v)}{S;\, v \,\&\&\, e \longrightarrow S;\, e}\ [\text{RT-And-Tru}]$$

$$\frac{\mathsf{falsy}(v)}{S;\, v \,\&\&\, e \longrightarrow S;\, v}\ [\text{RT-And-Fls}] \qquad \frac{\mathsf{truthy}(v)}{S;\, v \,||\, e \longrightarrow S;\, v}\ [\text{RT-Or-Tru}] \qquad \frac{\mathsf{falsy}(v)}{S;\, v \,||\, e \longrightarrow S;\, e}\ [\text{RT-Or-Fls}]$$

$$\frac{v' = \neg\mathsf{toBool}(v)}{S;\, !\, v \longrightarrow S;\, v'}\ [\text{RT-Neg}] \qquad \frac{\ell \text{ fresh} \qquad \mathrm{H}' = \mathrm{H},\, \ell \mapsto \{\mathtt{f}_1 : v_1, \ldots, \mathtt{f}_n : v_n\}}{S;\, \{\mathtt{f}_1 : v_1, \ldots, \mathtt{f}_n : v_n\} \longrightarrow S \triangleleft \mathrm{H}';\, \ell}\ [\text{RT-Record}]$$

$$\frac{S.\mathrm{H}(\ell) = \{\overline{\mathtt{f}_i : v_i},\, \mathtt{f} : v,\, \overline{\mathtt{f}_j : e_j}\}}{S;\, \ell.\mathtt{f} \longrightarrow S;\, v}\ [\text{RT-FldRd}] \qquad \frac{\mathrm{H}' = S.\mathrm{H}[\,\ell \mapsto S.\mathrm{H}(\ell)[\,\mathtt{f} \mapsto v\,]\,]}{S;\, \ell.\mathtt{f} = v \longrightarrow S \triangleleft \mathrm{H}';\, v}\ [\text{RT-FldWr}]$$

$$\frac{\mathrm{H}' = \mathrm{H}[\,\ell \mapsto v\,]}{S;\, \mathtt{var}\ x = v \longrightarrow S \triangleleft \mathrm{H}';\, \mathtt{skip}}\ [\text{RT-Var}] \qquad \frac{s \equiv \mathsf{truthy}(v)\ ?\ s_1\ :\ s_2}{S;\, \mathtt{if}\ (v)\ \{s_1\}\ \mathtt{else}\ \{s_2\} \longrightarrow S;\, s}\ [\text{RT-If}]$$

$$\frac{S.\mathrm{X} = \mathrm{X}',\, \mathrm{L.E} \qquad S' = S.\mathrm{H}; \mathrm{X}'; \mathrm{L}}{S;\, \mathtt{return}\ v \longrightarrow S';\, \mathrm{E}\langle v\rangle}\ [\text{RT-Ret}] \qquad \frac{}{S;\, \mathtt{skip}; s \longrightarrow S;\, s}\ [\text{RT-Skip}]$$

Fig. 11. Operational Semantics of FlowCore

5.1 Declarative Type System

This system assigns concrete types, *i.e.*, types stripped off of type variables, to language terms of FlowCore. Environments Γ also map program variables to concrete type entries (where both flow-sensitive and flow-insensitive types are concrete). The same holds for effects ε. The typing judgments for expressions and statements are: $\Gamma \Vdash e : \tau;\, \varepsilon;\, \psi \dashv\!\vdash \Gamma'$ and $\Gamma \Vdash s : \varepsilon \dashv\!\vdash \Gamma'$. The respective rules for these judgments follow the main intuitions of the inference system and are therefore deferred to the extended version [Chaudhuri et al. 2017, Section 5.1], along with other attendant definitions.

A *substitution* ρ maps type variables of the inference system to concrete types of the declarative system, and can be extended to types, effects and environments in a point-wise manner. Constraints c in the inference system correspond to subtyping relations in the concrete system for both types and effects. We can use the same substitution ρ to convert a constraint c to one or multiple subtyping

constraints over concrete types or effects. Since our type language has been kept simple overall, the subtyping rules for concrete types are routine and so a discussion is deferred to the extended version [Chaudhuri et al. 2017]. We say that a substitution ρ *satisfies* a constraint set C if all subtyping constraints generated by mapping ρ over C are valid. In this case we write $\rho \vdash C$.

We argue about the soundness of our type inference system with respect to the declarative system with the following lemma.

Lemma 5.1 (Soundness of Type Inference). *If* $\Gamma \vdash e : \tau ; \varepsilon ; \psi \dashv \Gamma' \triangleright C$ *and there exists substitution* ρ *s.t.* $\rho \vdash C$, *then* $\rho(\Gamma) \Vdash e : \rho(\tau) ; \rho(\varepsilon) ; \psi \dashv\!\Vdash \rho(\Gamma')$.

5.2 Type Safety

Before we state our type safety result for the declarative system, we extend the type checking judgment to runtime configurations: $\mathrm{G} \Vdash_{\Sigma} S; e : \tau$. Here G is a flow-insensitive environment mapping variables to their most general concrete type throughout the program. The judgment is to be read as: under a *heap typing* Σ, mapping heap locations to types and a flow-insensitive environment G, a configuration S; e is a assigned a type τ. We can now state our type safety result.

Theorem 5.2 (Type Safety). *For a configuration* S; e *and heap typing* Σ, *if* $\mathrm{G} \Vdash_{\Sigma} S; e : \tau$, *then:*

- ***(Preservation)*** *If* $S; e \longrightarrow S'; e'$, *then there exists* Σ', *such that* $\mathrm{G} \Vdash_{\Sigma'} S'; e' : \tau'$.
- ***(Progress)*** *Either* e *is a value, or there exists a configuration* $S'; e'$ *such that* $S; e \longrightarrow S'; e'$.

Supporting lemmas and proofs for the above results can be found in the extended version.

6 TYPE ANNOTATIONS

So far, we have described a system for *type inference* that ensures that values are used in ways that are consistent with their definitions. But can we check that the inferred types of values are consistent with types we specify?

In this section, we introduce a system for *type checking*. Type annotations τ follow a similar grammar as types except that there are no type variables, types can appear anywhere type variables could appear, and there are no effects. We consider a type annotation to be just another kind of type use, that expects some type of values. In other words, like everything else we can formulate type checking with flow constraints.

Technically, we need some new propagation rules for flow constraints involving type annotations.

When we see a constraint of the form $L \le \{\mathtt{f}_1 : \tau_1, \ldots, \mathtt{f}_n : \tau_n\}$, we propagate it with new constraints $L \le \mathrm{Get}(\{\mathtt{f}_1 : \alpha_1\})$, $\alpha_1 \le \tau_1$, $L \le \mathrm{Set}(\{\mathtt{f}_1 : \tau_1\})$, …, $L \le \mathrm{Get}(\{\mathtt{f}_n : \alpha_n\})$, $\alpha_n \le \tau_n$, $L \le \mathrm{Set}(\{\mathtt{f}_n : \tau_n\})$. (As mentioned Section 3.1, when L is a record type, these flow constraints can be replaced by unification constraints as an optimization.)

The remaining propagation rules rely on some new definitions.

Escaping effects. Since a function type annotation has nothing to do with any particular function expression, we cannot calculate its effect in the usual manner. Instead, we assume that there is an effect variable $\omega^\star$ that captures "escaping" effects, and that all function type annotations have this effect. When we see a constraint of the form $L \le \tau_1 \to \tau_2$, we propagate it with the new constraints $L \le \mathrm{Call}(\tau_1 \xrightarrow{\omega^\star} \alpha_2)$ and $\alpha_2 \le \tau_2$.

Conditional flow constraints. Checking that an inferred type is consistent with a union type annotation is tricky. Intuitively, this amounts to checking that the inferred type is consistent with either case of the union type annotation. But since the inferred type may contain type variables, it may not be obvious which case to pick. Consider the code in Figure 12 (left), where the parameter f on line 33 has a type that is the union of two function types. The call on line 33 is safe, since both

```
29 type IDString = (string) => string;
30 type IDNullableString = (?string) => ?string;
31 type Ambiguous = IDString | IDNullableString;
32
33 function onString(f: Ambiguous) { f(""); }
34 var id = (x) => x;
35 onString(id);
36 id(null);
```

```
37 type Correlated
38   = { type: "string", val: string }
39   | { type: "number", val: number };
40
41 function stringIsString(x: Correlated) {
42   if (x.type === "string")
43     displayString(x.val);
44 }
45 stringIsString({ type: "string", val: 0 });
```

Fig. 12. Type Annotations

function types take `string`. However, it is unclear which choice of function type to use for `id` on line 35. Picking `IDString` seems fine "locally," but turns out to be the wrong choice since **`null`** is passed on the next line and picking `IDNullableString` instead would type check.

Alternatively, we could propagate the choice further, effectively introducing disjunction in the logic of flow constraints. While appealing from a theoretical perspective, in practice this approach is complicated to implement and difficult to scale. It also doesn't mesh well with refinements. Consider the code in Figure 12 (right). Here, we propagate the choice of type for the object argument on line 45 into the fields of the object types in the union `Correlated`; this causes the call to type check, since the `type` and `val` fields of the object separately typecheck against the corresponding unions. However, this is unsound, as the call on line 43 shows (it passes a number at run time where a string is expected).

Instead, our approach is to try picking a case without ambiguity (*i.e.*, without considering type variables), or demand further type annotations to disambiguate. Specifically, let any constraint of the form $\alpha \leq \star$ or $\star \leq \alpha$ be a *condition*: its validity is conditional on what type α is inferred to be. Suppose that we have a restricted form of constraint propagation without the transitivity rule CP-Trans-T, so that conditions are not propagated. The propagation rule for choice, described below, uses this restricted form of constraint propagation to generate a set of constraints. If the rule signals an ambiguity, then the developer must provide annotations for the type variables involved in any generated conditions. Otherwise, the generated conditions are propagated further using the unrestricted rules.

A constraint of the form $L \leq \tau_1 \sqcup \tau_2$ is propagated as follows:

- Either $L \leq \tau_1$ generates an inconsistent set of constraints; then we continue with $L \leq \tau_2$.
- Or $L \leq \tau_1$ generates a consistent set of constraints c_1.
  - Either $L \leq \tau_2$ generates an inconsistent set of constraints; then we continue with c_1.
  - Or $L \leq \tau_2$ generates a consistent set of constraints c_2.
    - Either $c_1 \subseteq c_2$; then we continue with c_1.
    - Or $c_1 \not\subseteq c_2$; then we signal an ambiguity.

7 MODULES AND DEPENDENCIES

Until now, we have presented Flow's analysis on "whole" programs. However, JavaScript codebases can be quite large (*e.g.*, at Facebook we have millions of lines of JavaScript code), and a whole-program set-based analysis is simply not fast enough at scale.

In this section, we show how we modularize Flow's analysis. Modularization is important for performance, in terms of both time and space. It is also a natural fit for modern JavaScript, where code is typically split across a (large) number of (small) files; every file is mapped to a module, and possibly imports other modules; definitions are local by default, unless they are exported; and accessing global definitions other than builtins is generally discouraged.

Broadly, we follow the standard approach of analyzing each file separately, once all files it depends on have been analyzed. This strategy allows us to incrementally analyze the program as files change (Section 8), and parallelize the analysis across files (Section 9).

The key idea is to demand a "signature" for every module. We ensure that types inferred for the expressions a module exports do not contain type variables—wherever they do, we demand type annotations. For example, the parameter of an exported function expression must specify its type. Otherwise, we risk having the type of the parameter depend on calls in other files that import this module, which breaks modularization. (Alternatively, we could try generalizing the parameter's type based on how it is used inside the function expression, but in our experience it leads to unwieldy types.)

Requiring annotations for module interfaces is much better than requiring per-function annotations: a typical module exports one object or function, while having a bunch of module-internal code. The minimum annotation burden is only a small fraction of lines of code. (Of course, annotations are permissible even where they are not required.)

The type annotation syntax is designed to be nearly as expressive as the internal type language, but it is not true that *all* well-typed code can be cleanly refactored into modules to preserve typing. This is by design: modules are abstractions, and so some rewriting might be needed to prevent leaking abstractions. For example, effects do not appear in types at all, and local effects that leak after refactoring would need rewriting.

Independently, having a signature for every module turns out to be a desirable choice for software engineering. It is considered good practice for documentation (files that import the module can simply look up its signature instead of its implementation), as well as error localization (blames for errors do not cross module boundaries).

Modules, exports, and imports. For the purposes of this paper, let us assume that every file maps to a module by the same name. (This is sufficient to model the popular CommonJS module system, which is the default module system in Flow. However, we also support module systems where module names are independent of file names and not necessarily in 1-1 correspondence.)

A file can import definitions that another file exports. However, the reference m to the exporting file inside the importing file is *relative* to the importing file.

Module loading and dependency tracking. We assume a module loading judgment $\mathbb{FS} \Vdash F_i :: m \rightsquigarrow F_e^?$ that, given a file name F_i and a reference m, either computes the name F_e of the file being referenced by m in F_i or errors, while recording any files looked up by the derivation in $\mathbb{FS}$ (which may or may not exist). Whenever $\mathbb{FS} \Vdash F_i :: m \rightsquigarrow F_e$, we assume F_e exists and have F_e in $\mathbb{FS}$.

Given a file system state that satisfies the constraints $\mathbb{FS}$, we say that F_i *depends* on F_e whenever $\mathbb{FS} \Vdash F_i :: m \rightsquigarrow F_e$.

Compilation and linking. Files are "compiled" and "linked" in dependency order. In practice, there may be cycles in the dependency graph, so this process is run on the directed acyclic graph of strongly connected components, where each strongly connected component is considered to have all the imports and exports of the files in it. For each file, compilation and linking generates exported types and *signature* constraints of the form $\hat{\tau} \leq \alpha$, where any type variables in $\hat{\tau}$ are in positive positions. Such signature constraints fully describe the types of exports of the file, which can be "substituted" for the types of corresponding imports in dependent files.

Suppose that compiling a file F, with module references typed as fresh type variables $\alpha_1, \ldots, \alpha_n$, generates constraints c and an exported type τ. Furthermore, suppose that the module references resolve to files $F_1, \ldots, F_n$ that have their signature constraints $c_1^\star, \ldots, c_n^\star$ and exported types $\tau_1, \ldots, \tau_n$.

Then, we link F by adding $c_1^\star, \ldots, c_n^\star$ to c, propagating the constraints $\tau_1 \leq \alpha_1, \ldots, \tau_n \leq \alpha_n$, and transforming c to $c^\star$ by the process signature(τ), defined recursively as follows:

(1) signature(α) throws away upper bounds of α, and calls signature($\hat{\tau}$) for each $\hat{\tau} \leq \alpha$.
(2) signature($\hat{\tau}$) demands a type annotation for each type variable in a negative position in $\hat{\tau}$, and calls signature(α) for each α in a positive position in $\hat{\tau}$.

Intuitively, this process walks over constraints, with the exported types as roots, while doing two things. One, any constraints that would be unreachable when the exported types flow to type uses in dependent files are pruned away. Two, the developer must provide annotations wherever constraints could propagate back from dependent files.

Formally, a key property of signature constraints $c^\star$ and exported types τ is that it is impossible for a constraint of the form $\tau \leq u$ to lead to, via propagation, a constraint of the form $\hat{\tau} \leq \alpha$ where α is in $c^\star$. In other words, signature constraints and exported types can be considered "closed" when linking dependent files. This means that they do not need to be recomputed for correctness—in fact, they can be memoized and reused—which is crucial for performance. This also means that whenever "diamonds" occur in the dependency graph, *i.e.*, F depends on another file via multiple paths, the signature constraints of F are the same no matter which order the paths are explored.

8 INCREMENTALIZATION

In this section, we show how Flow exploits modularity and dependency management to incrementally analyze files as they change.

Architecture. Flow's architecture consists of a server, a client, and a file system watcher. The server initially analyzes the entire codebase, following the procedure in the previous section, and stores a bunch of information in memory. The information not only includes the status (type errors), but also the results of separately compiling and linking every file, and the dependencies between files. Once the server is initialized, it runs in the background.

The client queries the server for information, relaying commands issued via the command-line or various IDEs. Typically, the client is interested in the status. But the client could also ask for the type at a particular position, the definitions reaching a particular reference, *etc.*, in which case the server computes that information almost instantaneously from the information already stored in memory.

Finally, the file system watcher informs the server of changes to the file system: which files have been added, modified, or deleted. Based on this information, the server re-analyzes a (hopefully small) fraction of the code base, and updates the information stored in memory. (When the client queries the server again, the response is based on this updated information.)

Incremental analysis. Files that are added or modified need to be re-analyzed. In addition, a subset of unmodified "dependent" files needs to be re-analyzed. This set can be partitioned into *direct* dependents and *indirect* dependents.

Direct dependents are computed as follows. For any file F that is added, modified, or deleted, whenever F is in $\mathbb{FS}$ for any unmodified F_i where $\mathbb{FS} \Vdash F_i :: m \rightsquigarrow F_e^?$, we consider F_i is a direct dependent. The *depends* relation is modified by re-resolving module references in direct dependents. Indirect dependents are unmodified files that recursively depend on direct dependents.

9 PARALLELIZATION

We use a map/reduce algorithm augmented with shared memory communication for parallelizing various stages of type checking, building on and extending Hack's model [Hack 2014].

9.1 Workers

Assume we want to distribute a task on a number of files, computing a result of type R. We describe the task with a function `job` : *Files* → W; a function `merge` : $R \times W \to R$; a value `neutral` : R; and a function `next` : () → *Files*.

We have a master process and as many worker processes as the number of available processors. The master initially has *result* as `neutral`, and considers all workers free. Then, it repeatedly does the following:

- If there is a free worker, call `next`() to obtain a list of files *files*. If *files* is empty and all workers are free, exit with result *result*. If *files* is not empty, send it to the free worker.
- If a worker has sent back w, consider the worker free, run `merge`(*result*, w) and update *result* with it.

Correspondingly, every worker repeatedly does the following. If the master has sent *files*, fork a process to run `job`(*files*), wait for a value w, and send back w to the master.

Usually, the `next` function is simple. It just remembers an index into the original list of files. When the index is out of bounds, it returns an empty list; otherwise it returns a sublist from that index of some fixed "bucket" size, and advances the index. This models processing a static list of files in no particular order.

What if the processed list is computed dynamically, in a particular order? For that, we make the following changes. We maintain a "worklist" of files, and have `next` create a bucket from the worklist instead. Let the intermediate result type W' be *Files* × W. Then `job`′(*files*) returns (*files*, `job`(*files*)). Finally, `merge`′(*result*, (*files*, w)) updates the worklist with *files*, before returning `merge`(*result*, w).

9.2 Shared Heap

As described above, the master and the workers communicate by serializing and deserializing data like files and intermediate results. In practice, the results that need to be computed are often maps from files to large values, and thus communicating intermediate results from the workers back to the master becomes a performance bottleneck. Furthermore, manipulating large results in the master spikes its memory usage and causes frequent garbage collection pauses, which affects the server's responsiveness.

Thus, we use a different mechanism for sharing results: a large hashtable mapped to RAM, accessible to both the master and the workers, that is logically divided into various maps. The hashtable provides fast (lock-free) concurrent access for reads, as well as for writes as long as they *add* entries for *disjoint* keys. Only the master can remove entries. These conditions turn out to be easily satisfied in our setting: at any stage of type checking, different workers always operate on different files, and old entries are only ever cleared to make way for new entries when processing file system changes in the master.

Entries are compressed on writes. (We use LZ4 [Collet 2011] because it is extremely fast, while providing sufficient compression.) In practice, this means we can tolerate redundancy in the entries, trading off space for time by precomputing information. Moreover, entries are cached on reads.

With the shared heap, the types R and W can be quite small (typically, metadata for bookkeeping). Moreover, the processes forked by workers to run jobs are short-lived: their memory is reclaimed by killing them on every completion.

9.3 Parallelizing Parsing

Files are parsed in parallel, in no particular order, using a static `next`. Every `job` writes the abstract syntax trees for corresponding files to shared memory.

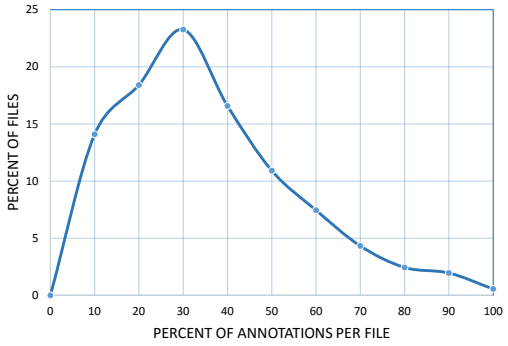

Fig. 13. Distribution of annotations

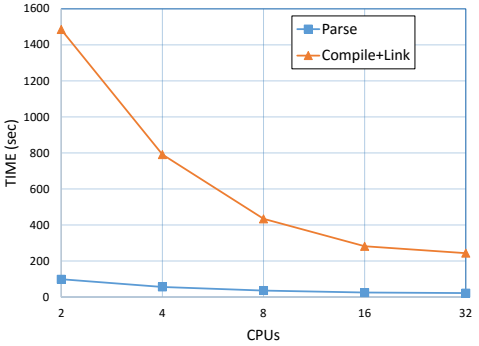

Fig. 14. Effects of parallelization on performance

9.4 Parallelizing Compilation and Linking

Next, the files are compiled and linked in parallel, following dependency order, using a dynamic `next`. We maintain a count of dependencies for each file. The worklist initially contains files that do not depend on any other files. As files are processed, they decrement the dependency counts of other files that depend on them, possibly causing those files to be added to the worklist (because they no longer depend on any files that have not already been processed).

Every `job` reads the abstract syntax trees for the corresponding files, and the signature flow constraints for the files they depend on, does compilation and linking, and finally writes the signature flow constraints for the corresponding files to shared memory.

Note that a file does not need to be rechecked if the signature flow constraints for none of its dependencies change. This is a major optimization: it means that even though a file may have a lot of recursive dependents, only a small fraction of them may actually need to be rechecked. Indeed, in practice it results in order-of-magnitude differences in recheck times. The implementation is slightly tricky because new signature flow constraints often contain fresh type variables that make them trivially different than old signature flow constraints. We compute and compare hashes modulo such trivial differences to detect when signature flow constraints have changed.

Compiling and linking files in dependency order limits some parallelism in theory, but in practice, the alternative approach of processing every file independently ends up doing far more work. Overall we save processing time by an order of magnitude.

10 EXPERIMENTS

We ran experiments on the main internal repository at Facebook, in which (at the time of writing this paper) around 13M LOC of JavaScript are covered by Flow, spanning around 122K files. We chose this repository because it contains a wide variety of JavaScript projects, implementing client code and frameworks for web applications, that depend on each other but are owned by different teams and do not necessarily conform to a uniform coding style. (A smaller repository contains client code and frameworks for mobile applications, which is also covered by Flow, but which we do not consider here—the general conclusions about Flow's behavior, however, remain the same.)

Distribution of annotations. For each file, we counted the number of annotations as a fraction of the total number of locations that could potentially be annotated. Figure 13 shows the distribution of these relative numbers across the repository. The median is 29% annotations. In aggregate, there are around 686K annotations vs. 1502K other locations that could potentially be annotated but

are not. Note that these numbers do not account for type inference of every expression—only declaration sites are considered—and thus conservatively overapproximate the annotation burden.

Usage of refinements. As a quick test of the impact of supporting refinements in Flow, turning it off led to more than 145K spurious errors across this repository.

Effects of parallelization on performance. Total initialization time was around 225s with 32 processors (Xeon, 2.2GHz), which means around 3 ms per file on average. We also used around 4GB of shared memory.

Figure 14 shows how performance varies with the number of processors (we only measured performance for powers-of-2 processors and extrapolated). Parsing times become half from 2 to 4 processors, but then the improvements slow down (since parsing is already quite fast, and communication starts dominating—an example of Amdahl's law). Compilation and linking times hit the limit much later, continuing to improve almost linearly until 16 processors. (This makes sense, since compiling and linking is much slower than parsing.)

Effects of code size on performance. The time taken to compile and link a file grows approximately linearly with code size. While set-based analysis is well-known to be worst-case cubic in code size, the lines of code per file is small (average 106), and the imports are summarized by signature constraints, which are much smaller than the corresponding lines of code of dependencies.

Effects of dependencies on performance. The time taken to process a file is not correlated with the number of recursive dependencies—the spread of times remains relatively flat as dependencies grow. This is not very surprising, since signature constraints of imports are supposed to "compress" the information contained in recursive dependencies. On the other hand, the time taken to process a file grows approximately linearly with the number of imports.

The size of the signature constraint graph, which form the results of compiling and linking, is not correlated with code size—for most files, the sizes are between 50-100KB.

Effects of incrementalization on performance. The performance of incremental type checking is tied to the number of files that are rechecked when a file is modified, *i.e.*, the number of files that recursively depend on that file.

The distribution of the number of recursive dependents is highly positively skewed—the median is less than 10, and the 90^{th} percentile is less than 100. Considering the time taken to recheck to be roughly proportional to the number of files to link, this means that in 90% of cases, recheck time is less than 200ms plus some constant. Of course this is only a rough calculation: in practice, while this closely approximates the common scenario of editing single files, it doesn't account for occasional rebases that can cause larger numbers of files to be rechecked.

11 RELATED WORK

There has been a lot of work on type systems for JavaScript and related languages, as well as set-based inference techniques. We focus here only on the most closely related work.

11.1 Mainstream Type Systems for Dynamic Languages

TypeScript [2012] is a widely used typed superset of JavaScript. Like Flow, it aims to improve developer productivity by providing tooltips through IDEs. Unlike Flow, it focuses only on finding "likely errors" without caring about soundness [Bierman et al. 2014]. Type inference in TypeScript is mostly local and in some cases contextual; it doesn't perform global type inference like Flow, so in general more annotations are needed. Whenever type annotations are missing, they are considered

to be any (instead of being implicitly inferred). Thus, many type errors are missed. Consider for example the program:

```
function square(n) { return n * n; }
square("oops");
```

Flow will signal an error for trying to use a string as an argument to a multiplication. TypeScript, on the other hand, will infer any as the type for n and accept this program as valid. To get a similar behavior from TypeScript we would need to add a type annotation to the parameter n of square.

Furthermore, even with fully annotated programs, TypeScript misses type errors because of unsound typing rules. For example, "bivariant" subtyping means that functions and instances of polymorphic classes can be passed to contexts that do not preserve their typing invariants, as can be seen in the following erroneous example:

```
var assertString = (x: string) => assert(typeof x === "string");
var app = (f: (x: string | number) => void, x: number) => f(x);
app(assertString, 1);
```

A checker that implements sound contra-variant argument subtyping, like Flow, would signal an error at the call to app, since assertString is incompatible in its argument with the expected type for parameter f of app. In practice, this means that TypeScript developers have to code defensively with dynamic checks, even when types are included. Safe TypeScript [Rastogi et al. 2015] "fixes" soundness problems in TypeScript with stricter typing rules and runtime enforcement mechanisms to restore gradual typing.

Dart [2011] is another language that shares the same philosophy. Unsoundness is a deliberate choice in TypeScript and Dart, motivated by the desire to balance convenience with bug-finding. But we have enough anecdotal evidence from developers at Facebook that focusing on soundness is not only useful but also desirable, and does not necessarily imply inconvenience. Similar to Safe TypeScript, recent work recovers soundness in Dart [Heinze et al. 2016].

Closure [2009] is another widely used type system for JavaScript that focuses on transforming code for size reduction. As far as we can tell, it is sound modulo similar assumptions as Flow, but lacks type inference. Typed Racket [Tobin-Hochstadt and Felleisen 2008], and Hack [2014] (for PHP) are also quite close in spirit: their optional typing is at the level of modules and they use occurrence typing to perform similar kinds of refinements. They differ in that they lack type inference and, compared to Flow, their treatment of mutable variables is far more simplistic—there is no distinction between mutability on the stack and on the heap. On the other hand, Flow heavily borrows from Hack's design and implementation for scaling to millions of lines of code.

11.2 Research Static Analysis for JavaScript

Early work. Several static typing systems have successfully been ported to the dynamic setting of JavaScript. Early work by Thiemann [2005] and Anderson et al. [2005] focus on restricted subsets of the language.

Type Refinement. In the area of type refinement, Guha et al. [2011] develop *flow typing*, which, like Flow, supports type narrowing as a consequence of control flow. Unlike Flow, their analysis is strictly intra-procedural, does not perform type inference and does not track non-local effects (*e.g.*, variable updates). Building on this work, Lerner et al. [2013] present a framework for building type systems for JavaScript, engineered modularly to encourage experimentation, but which also suffers from limited type inference compared to Flow.

Static Objects. Choi et al. [2015] propose a static type system for ahead-of-time compilation of JavaScript that guarantees fixed object layout. Its type inference is based on very similar foundations as Flow. SJS focuses mainly on taming legacy object-oriented features (constructor functions,

open methods, and prototype inheritance). While Flow does support these features, its model and guarantees are different—it models these features with *extensible* objects, and guarantees type consistency of gets and sets of properties (but necessarily their existence). Many of these concepts are replaced by classes in "modern" (ES6+) JavaScript, where Flow can provide stronger guarantees with advanced type system features like bounded polymorphism, `this` types, and read-only properties. Chandra et al. [2016] build on this work by adding support for abstract objects, first-class methods, and recursive objects, and prove their extensions sound. Their type system supports additional features such as polymorphic arrays, operator overloading, and intersection types in manually-written interface descriptors for library code, that they found important for building GUI applications. Their formalization focuses on their object model. Unlike Flow, they do not discuss type refinement based on conditional checks, and their formalization is flow-insensitive, so less precise in that respect compared to Flow.

Abstract Interpretation. This is a more heavyweight approach in statically analyzing JavaScript. Approaches here include TAJS [Andreasen and Møller 2014; Jensen et al. 2011, 2009, 2010], JSAI [Kashyap et al. 2014] and SAFE [Lee et al. 2012; Park and Ryu 2015]. These approaches vary in precision, user customizability and flow-, context- and path-sensitivity, but being whole-program analyses, they are out of scope at our scale, while being much more precise and not needing annotations.

Program Logics. Recent advances in SMT solver technology has spurred the interest in using program logics to track the dynamic behavior of JavaScript programs. DJS [Chugh et al. 2012] combines nested refinements with alias types [Smith et al. 2000], a restricted separation logic, to account for aliasing and flow-sensitive heap updates to obtain a static type system for a large portion of JavaScript. DJS comes with very limited type inference and hence requires complex annotations at function and loop boundaries. To reduce this annotation burden, Vekris et al. [2016] offer refinement type inference for TypeScript, based on Liquid Type inference [Rondon et al. 2008]. Their type system is more expressive than Flow's as it allows logical predicates (taken from a number of decidable logics) to be attached on a base type system (a subset of TypeScript's type system). However, this precision comes at the cost of a higher annotation burden and a penalty on scalability. Also, while offering global refinement type inference, unlike Flow it does not infer the base (underlying) types of programs and requires explicit immutability annotations.

11.3 Inference and Subtyping in Dynamic Languages

Constrained Types. Set constraints have been used for the purpose of type inference by Aiken and Wimmers [1992] and Aiken et al. [1994], who adopt the set-theoretic model to infer types in a simple functional language. Trifonov and Smith [1996] and Pottier [1998] infer polymorphic recursively constrained types, but retain a simpler interpretation of type terms. In their work, ground types are regular terms, and subtyping is defined explicitly on terms. This enables various simplifications to their constraint sets, like garbage collection [Eifrig et al. 1995; Pottier 1998, 2001]. Flanagan and Felleisen [1999] use a simpler type representation and, based on simplification algorithms that exploit the observable equivalence of constraint sets, perform *componential* set-based analysis.

Flow builds directly on work by Pottier [2001], but does not infer polymorphic types. Instead, it exposes features less frequently addressed in the context of set-constraint based analyses, such as variable updates and type refinement based on conditional checks. In addition, Flow's analysis is not context-sensitive, due in part to anecdotal concerns about performance in DoctorJS [Vardoulakis 2012]. In practice, polymorphic type annotations recover context-sensitivity where needed.

Constraint Graph Simplification. The research directions above already include several simplification techniques [Fähndrich and Aiken 1996; Flanagan 1997]. To further improve performance of

inclusion constraint analyses, Fähndrich et al. [1998] propose a technique for eliminating cycles in constraint graphs that is based on a non-standard graph representation called *inductive form*, and only traverses part of the paths during the search for cycles. To address the problem of redundant paths in a constraint graph, Su et al. [2000] propose *projection merging*, a technique intended to be used in conjunction with the above. In contrast, we directly implement unification constraints using union-find over a base representation of inclusion constraints.

Semantic and Algebraic Subtyping. Advances in semantic and algebraic foundations have spurred renewed interest in this rich area. Semantic subtyping has been proposed in the context of functional languages for XML based programming [Frisch et al. 2008], ML-like languages [Castagna et al. 2016], and more recently for imperative object-oriented languages, where fields can be mutable [Ancona and Corradi 2016], and in a gradual typing setting [Castagna and Lanvin 2017]. Even though polymorphic type inference with subtyping is known to be undecidable [Su et al. 2002], Dolan and Mycroft [2017] infer compact principal types by keeping a strict separation between the types used to describe inputs and those used to describe outputs (polarities). In comparison, Flow is less ambitious with union and intersection types.

12 LIMITATIONS AND THREATS TO VALIDITY

We conclude this paper by discussing limitations and threats to validity.

Flow's analysis is cubic in the worst case. Although pathological examples are not entirely uncommon, we have so far been able to mitigate them with low-hanging optimizations.

Its analysis is context-insensitive, and also not well-suited for libraries with reflection. Many libraries provide annotations without checked implementations, so we can typecheck the vast majority of code that uses these libraries. Better techniques for checking libraries (e.g., TAJS) can complement Flow.

Like many other type systems for dynamically typed languages, Flow has the `any` type, with which type checking can be completely bypassed. Unlike gradual type systems, though, there is no runtime enforcement of types when they interact with `any`. For sound gradual typing, the subtyping rules can be augmented to mark all type constructors as either trusted or untrusted.

Even without `any`, some aspects of JavaScript force us into choosing unsoundness where it is objectively justified. We can lay down the conditions for soundness, but not enforce them. For example, arrays in JavaScript can have "holes": it is possible to add an element out of bounds, in which case any intermediate positions are filled with `undefined`. Likewise, records in JavaScript can also be accessed as dictionaries, so it is possible to read and write a named property by passing a computed string. Short of complicated numeric and string analysis, soundness would demand that we lose type information on array dereferences and dictionary reads, but this is too restrictive in practice. Instead we hope that developers who care about soundness will not create arrays with holes (e.g., by always using `Array.push` to add elements), or will check for `undefined` on dereferences when needed; and the properties that are named and those that are accessed via computed strings are disjoint.

ACKNOWLEDGMENTS

Thanks to Basil Hosmer, Jeff Morrison, Nat Mote, Satish Chandra, Caleb Meredith, and James Kyle for their contributions to Flow's design and implementation, to Julien Verlaguet, Dwayne Reeves, and Yoann Padioleau for their work on infrastructure that Flow is built on, and to anonymous reviewers for their valuable feedback on previous drafts of this paper.

REFERENCES

Alexander Aiken and Edward L. Wimmers. 1992. Solving systems of set constraints. *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science* (1992), 329–340. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=185545

Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. 1994. Soft Typing with Conditional Types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. ACM, New York, NY, USA, 163–173.

Davide Ancona and Andrea Corradi. 2016. Semantic subtyping for imperative object-oriented languages. *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2016*, 568–587.

Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Towards Type Inference for Javascript. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*. Springer-Verlag, Berlin, Heidelberg, 428–452.

Esben Andreasen and Anders Møller. 2014. Determinacy in Static Analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 17–31.

Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28 – August 1, 2014. Proceedings*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 257–281.

Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. In *Proceedings of ICFP*.

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. 2016. Set-theoretic Types for Polymorphic Variants. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 378–391.

Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. 2016. Type Inference for Static Compilation of JavaScript. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 410–429.

Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and Precise Type Checking for JavaScript (Extended version). https://arxiv.org/abs/1708.08021. (2017).

Wontae Choi, Satish Chandra, George C. Necula, and Koushik Sen. 2015. SJS: A Type System for JavaScript with Fixed Object Layout. In *SAS (Lecture Notes in Computer Science)*, Vol. 9291. Springer, 181–198.

Ravi Chugh, David Herman, and Ranjit Jhala. 2012. Dependent Types for JavaScript. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 587–606.

Closure. 2009. Closure Compiler. https://developers.google.com/closure/compiler/. (2009). Accessed: 2016-11-15.

Yann Collet. 2011. LZ4-Extremely fast compression. (2011). https://github.com/lz4/lz4.

Dart. 2011. Dart Language Specification. https://www.dartlang.org/guides/language/spec. (2011). Accessed: 2016-11-15.

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages - POPL 2017*. ACM Press, New York, New York, USA, 60–72.

Jonathan Eifrig, Scott Smith, and Valery Trifonov. 1995. Sound Polymorphic Type Inference for Objects. *SIGPLAN Not.* 30, 10 (Oct. 1995), 169–184.

Manuel Fähndrich and Alex Aiken. 1996. *Making Set-Constraint Program Analyses Scale.* Technical Report. Berkeley, CA, USA.

Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. 1998. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. ACM, New York, NY, USA, 85–96.

Cormac Flanagan. 1997. *Effective Static Debugging via Componential Set-Based Analysis.* Ph.D. Dissertation. Rice University.

Cormac Flanagan and Matthias Felleisen. 1999. Componential Set-based Analysis. *ACM Trans. Program. Lang. Syst.* 21, 2 (March 1999), 370–416.

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic Subtyping: Dealing Set-theoretically with Function, Union, Intersection, and Negation Types. *J. ACM* 55, 4, Article 19 (Sept. 2008), 64 pages.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2011. Typing Local Control and State Using Flow Analysis. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software (ESOP'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 256–275.

Hack. 2014. Hack Language Specification. https://github.com/hhvm/hack-langspec. (2014). Accessed: 2016-11-15.

Thomas S. Heinze, Anders Møller, and Fabio Strocco. 2016. Type Safety Analysis for Dart. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS 2016)*. ACM, New York, NY, USA, 1–12.

Simon Holm Jensen, Magnus Madsen, and Anders Møller. 2011. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 59–69.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5673 LNCS (2009), 238–255.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2010. Interprocedural Analysis with Lazy Propagation. In *Proceedings of the 17th International Conference on Static Analysis (SAS'10)*. Springer-Verlag, Berlin, Heidelberg, 320–339.

Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 121–132.

Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, Vol. 10.

Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. 2013. TeJaS: Retrofitting Type Systems for JavaScript. *Proceedings of the 9th symposium on Dynamic languages - DLS '13*, 1–16.

Changhee Park and Sukyoung Ryu. 2015. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 735–756.

François Pottier. 1998. *Type Inference in the Presence of Subtyping: from Theory to Practice.* Research Report 3483. INRIA. http://hal.inria.fr/docs/00/07/32/05/PDF/RR-3483.pdf

François Pottier. 2001. Simplifying Subtyping Constraints: a Theory. *Information & Computation* 170, 2 (Nov. 2001), 153–183.

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 167–180.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 159–169.

Frederick Smith, David Walker, and J. Gregory Morrisett. 2000. Alias Types. In *Proceedings of the 9th European Symposium on Programming Languages and Systems (ESOP '00)*. Springer-Verlag, London, UK, UK, 366–381.

Zhendong Su, Alexander Aiken, Joachim Niehren, Tim Priesnitz, and Ralf Treinen. 2002. The First-order Theory of Subtyping Constraints. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM, New York, NY, USA, 203–216.

Zhendong Su, Manuel Fähndrich, and Alexander Aiken. 2000. Projection Merging: Reducing Redundancies in Inclusion Constraint Graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*. ACM, New York, NY, USA, 81–95.

Peter Thiemann. 2005. Towards a Type System for Analyzing Javascript Programs. In *Proceedings of the 14th European Conference on Programming Languages and Systems (ESOP'05)*. Springer-Verlag, Berlin, Heidelberg, 408–422.

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 395–406.

Valery Trifonov and Scott Smith. 1996. Subtyping Constrained Types. *SAS '96: Proceedings of the 3rd International Symposium on Static Analysis* (1996), 349–365.

TypeScript. 2012. TypeScript Design Goals. https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals. (2012). Accessed: 2016-11-15.

Dimitris Vardoulakis. 2012. *CFA2: Pushdown Flow Analysis for Higher-Order Languages.* Ph.D. Dissertation. Northeastern University.

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement Types for TypeScript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 310–325.

A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (Nov. 1994), 38–94.