Contents

1 Introduction

Flow is a static type system for JavaScript, created and maintained by Facebook. In this presentation, we focus specifically on how Flow manages **type refinements** and invalidations thereof, particularly in the context of *mutable local variables* captured by reference in closures.

2 Example code

Consider the following JavaScript code, which typechecks just fine in Flow (you can try it here):

```
var robby = { name: 'Robby'};

function getName(x) {
  x = x || robby;
  return x.name;
}
```

In the function `getName`, the line `x = x || robby;` is a common JavaScript idiom, and assigns `x` a default value of `robby`. It's equivalent to `x = x ? x : robby`.

However, the following function *fails* to typecheck:

```
var robby = { name: 'Robby'};

function getNameFail(x) {
  x = x || robby;
  function reset() { x = null; }
  reset();
  return x.name;  // ERROR: Cannot get `x.name` because property `name` is missing in null
}
```

The added lines `function reset() { x = null; }` and `reset();` declare and invoke, respectively, a closure `reset` which captures a reference to the local argument `x`, and resets it to `null`.

Flow is unhappy with this, and gives the following error:

```
15:   return x.name;
                ^ Cannot get `x.name` because property `name` is missing in null [1].
References:
13:   function reset() { x = null; }
                              ^ [1]
```

Flow recognizes that invoking `reset()` sets `x` to type `null`, so the lookup `x.name` will fail.

This is an example of **flow-sensitive** analysis – Flow traces the program's dataflow, generating and propagating constraints on types and variables at each step. If a contradiction is reached, Flow will give a type error.

3 Typing judgments

The core typing judgment for expressions in Flow is of the form

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

which should be read as "Environment Γ proves that e has type τ, *and* evaluating e has **effects** ε, *and* when e is truthy we know **predicates** ψ. This judgment produces an **output environment** Γ', and a corresponding **constraint set** C."

Statements have a strictly simpler typing judgment:

$$\Gamma \vdash s : \varepsilon \dashv \Gamma' \triangleright C$$

which is identical to the judgment for expressions, except we don't prove a type (because statements can't have types) and we don't learn any predicates (because statements can't be truthy).

There are several new concepts here, and we'll illustrate them below in the inference rules. As a whirlwind tour, however:

- **Effects** (represented with ε, ω) are program variables, like `x` or `y`. An effect ε associated with some expression e roughly describes the set of variables mutated by evaluating e.
- **Predicates** (represented with $\psi_1, \psi_2, \ldots$) are statements like $x \mapsto$ truthy. A predicate P maps a variable to what we know about its type. In the typing judgment above, a predicate set ψ associated with some expression e is roughly the additional typing information we learn when e is truthy.
- **Environments** (represented with $\Gamma_1, \Gamma_2, \ldots$) map
- **Constraints** (represented with $C_1, C_2, \ldots$) are the trickiest concept to understand

4 Constraint generation

4.1 Record literals: CG-Rec

The inference rule CG-Rec says that an environment Γ proves that expression $\{f : e\}$:

- Has **type** $\{f : \alpha\}$, where α is a fresh type variable;
- With the **effects** ε of evaluating e;
- No new **predicates**;
- And outputs:
  - **Environment** Γ', the output of evaluating e via the standard judgment $\Gamma \vdash e : \tau; \varepsilon; \psi \dashv \Gamma' \triangleright C$, and
  - **Constraint** $C \cup \{\tau \leq \alpha\}$, the union of the constraints from evaluating e and a new constraint that says e's type τ is a subtype of field f's type α in the result.

For the example code

```
var robby = { name: 'Robby' };
```

this is equivalent to writing, for some type variable `Alpha`, the following type assertions:

```
(robby: { name: Alpha });
('Robby': Alpha);  // 'Robby' <: Alpha
```

4.2 Variable assignments: CG-Assign

The previous rule actually only evaluates `{ name: 'Robby' }`. To evaluate the full assignment

```
var robby = { name: 'Robby' };
```

we use rule CG-Assign.

4.3 Function declarations: CG-Fun

```
function getName(x) {
    // ...s
    return e;
}
```

A successful typecheck for $(x) \to \{s; \text{return } e;\}$ tells us:

- The function has type $\alpha \xrightarrow{\varepsilon \setminus x, \overline{x_i}} \tau$, where
  - α is a fresh type variable created for the parameter,
  - $\varepsilon \setminus x, \overline{x_i}$ describes the effect had by calling the function later.
    - ∗ This is exactly the effect ε of evaluating the function's body, but we remove the parameter x and local variables $\overline{x_i}$, since those are local to the function and their invalidation is irrelevant to the calling context.
  - τ is the evaluated type of e, the return value in the function body.
- No new effects, denoted $\bot$ – just declaring a function doesn't invalidate anything.
- No new predicates, denoted $\emptyset$. Again, just declaring a function doesn't tell us anything new about the types of our variables.

4.4 Variable references: CG-Var

```
robby
```

Looking up a variable in our environment, Γ, tells us that `robby`

- Has a type, τ^{α}
- results in no effects
- Gives us the predicate that `robby` is truthy
- In the same environment
- With no new constraints.

4.5 Logical OR: CG-Or

```
x || robby
```

This expression can evaluate to one of two things: either 1) `x`, if `x` is truthy or 2) `robby`, if `x` is falsey. This is reflected in the rule for CG-Or, which tells us that:

- The resulting expression has type $\alpha \cup \tau_2$, where
  - α is a fresh type variable and
  - tau_2 is the type which results from evaluating `robby`. tau_2 is proved by Γ_1', which is produced through a refinement on Γ_1, the environment which results from evaluating `x`.
- The resulting effect is $\varepsilon_1 \cup \varepsilon_2$, where
  - ε_1 are the effects associated with evaluating x and
  - ε_2 are the effects associated with evaluating $robby$.
- The predicate ψ is produced, which is the result of a logical or on
  - $\psi_1 \setminus \varepsilon_2$, the predicates associated with `x` without the effects associated with`robby` and
  - ψ_2, the predicates associated with `robby`
- We get a new environment, Γ', which comes from the union of:
  - Γ_1'', the result of refining the environment obtained after evaluating `x` with ψ_1, the predicates associated with `x`
  - Γ_2, the environment obtained after evaluating `robby`
- The new constraints are the union of:
  - C_1, the constraints from evaluating `x`,
  - C_2, the constraints obtained from applying the negation of predicates ψ_1 on Γ_1 (the contraints of `x` being falsey),
  - C_3, the constraints from evaluating `robby`,
  - C_4, the constraints obtained from applying the predicates ψ_1 on Γ_1 (the constraints of `x` being truthy), and
  - and a new constraint that tau_1, the type of `x`, flows to the boolean value of `x`

4.6 Record lookup: CG-FdRd

```
x.names
```

4.7 Function calls: CG-Call

5 Refinements

5.1 Applying refinements: Ref-Single

5.2 Invalidating refinements: Ref-Effect

6 Constraint propagation

6.1 Variable subtyping transitivity

6.2 Function calls: CP-Call

6.3 Refinement effects: CP-Havoc

6.4 Record lookup: CP-Get

7 Putting it all together: derivation

7.1 Consistency

7.2 Derivation

8 Citations