Case Study

Zheng Gao, Christian Bird, Earl T. Barr

October 2016

Based on four criteria, we select bugs for further manual assessment: ones whose TypeScript- or Flow-preventability is not agreed upon, ones whose TypeScript- or Flow-preventability remains unknown in the assessment stage, ones whose TypeScript- and Flow-preventability differ, and ones that are TypeScript-preventable under version 2.0 but not under 1.8.

Disagreements Of the 80 uniformly-sampled bugs that we used to calculate inter-rater agreement, each rater needed to make 160 decisions in total, 80 for TypeScript-preventability and 80 for Flow-preventability. 138 of these 160 decisions were unanimously labelled. We define a *strong* disagreement as a disagreement in which one rater deems the bug preventable while another deems it unpreventable. Of the 22 disagreements, 12 are strong.

Let U denote unknown, P preventable, and $\overline{P}$ unpreventable. We manually assessed each disagreement without a time bound and found that, in each case, weak disagreements resolved as follows: $UUP \rightarrow P, UU\overline{P} \rightarrow \overline{P}, UPP \rightarrow P, U\overline{PP} \rightarrow \overline{P}$. In other words, the rater who confidently assessed *ts*-(un)preventability within the time bound was correct every time in our experiment. Our 12 strong disagreements had three patterns of labels: 2 were $P\overline{P}U$, 2 were $\overline{P}\overline{P}P$, and 8 were $PP\overline{P}$. After manually resolving all of them, we found that whenever two raters agreed, they were correct. Among the 10 strong disagreements where a rater disagreed with the other two, rater one dissented in 8 cases and rater two in 2 cases. With hindsight, we would have improved our assessment protocol. We should have specified that each rater consider whether or not added logic was manual type checking. We would have agreed on whether or not to consider typos in library names *ts*-preventable. These changes alone would have eliminated 7 of the 12 strong disagreements. Please visit our project page for more details.

Unknowns For the 320 bugs outside the inter-rater dataset, we could not always decide whether they are *ts*-preventable within 10 minutes, leaving 18 unknown. The main obstacles we encountered during the assessment include complicated module dependencies, the lack of fully annotated interfaces for some modules, large, tangled fixes that prevented us from isolating the region of interest, mismatches between the error models of Flow and TypeScript, and the general difficulty of program comprehension.

For these 18 bugs, we spent as long as needed until we resolved all the unknowns.

We patiently imported all relevant modules by using interface management tools like Typings[1], wrote our own annotated interfaces as appropriate, and read the code base and official documentation when necessary. To validate a *ts*-unpreventable assessment, we devised a simple experiment to validate it, as necessary.

As a result of this work, we have successfully labelled all 400 bugs as either preventable or unpreventable under TypeScript and Flow. TypeScript could have prevented two of these unknowns for a grand total of 58 (14.5%); Flow catches one more and reaches 60 (15%) in total. The updated binomial test results shows that at the confidence level of 95%, the true percentage of Flow- and TypeScript-preventable bugs falls in the range of $[11.5\%, 18.5\%]$ and $[11.1\%, 18\%]$.

The time spent assessing each of these 18 bugs varied significantly, ranging from 8 minutes to more than 1 hour of dedicated time. Surprisingly, 3 bugs took us only around 10 minutes to decide their *ts*-preventability on a fresh restart, which, we reckon, is due to our increasing expertise.

Classifying ts-unpreventable Bugs Figure 1 categorizes bugs that are unprevenatble under both Flow and TypeScript, after the 18 unknowns were resolved. Recall that, while `BranchError`, `PredError`, and `URIError` are logic errors in implementing the specification, `SpecError` captures all other specification errors. Unsurprisingly, `SpecError`, with 186 bugs, accounts for 55% of the total bugs and significantly outweighs other categories. Errors implementing specification, as a group, overwhelmingly constitute 78%. This result, yet again, demonstrates the importance of specifications.

Despite the dominance of errors implementing specification and the fact that only public bugs are considered, there still exists a non-specification-related opportunity for type system: `StringError`. Ranked second in the histogram, `StringError` is a broad concept that represents errors caused by the incorrect content of a string, such as a wrong URL. The reason why `StringError` is so common, we conjecture, is two-fold: first, the `string` type itself is extremely popular; second, JavaScript is rooted in web applications that extensively use hyperlinks. However, the `string` type is opaque to most static type systems, and how to effectively refine it remains challenging.

Measuring TypeScript 2.0 null Handling Improvement TypeScript 2.0 was released during this study, giving us the opportunity to measure the effectiveness of its handling of `null` and `undefined`. Prior to 2.0, all types were nullable in TypeScript. In Flow, all types, except `any`, `void`, and `null`, are non-nullable by default; one can designate them as nullable by prefixing them with `?`. This design choice enables Flow to elegantly catch incorrect `null` / `undefined` usage. TypeScript 2.0 added the compiler option `--strictNullChecks`, which, when enabled, makes most types nonnullable, while allowing the user to specify nullability by or-ing `null` into a type annotation. For instance, `var s: string | null = "foo";` defines `s` to be a nullable string.

We reviewed our corpus and found that 22 bugs, an increase of 38%, are

[1] https://github.com/typings/typings

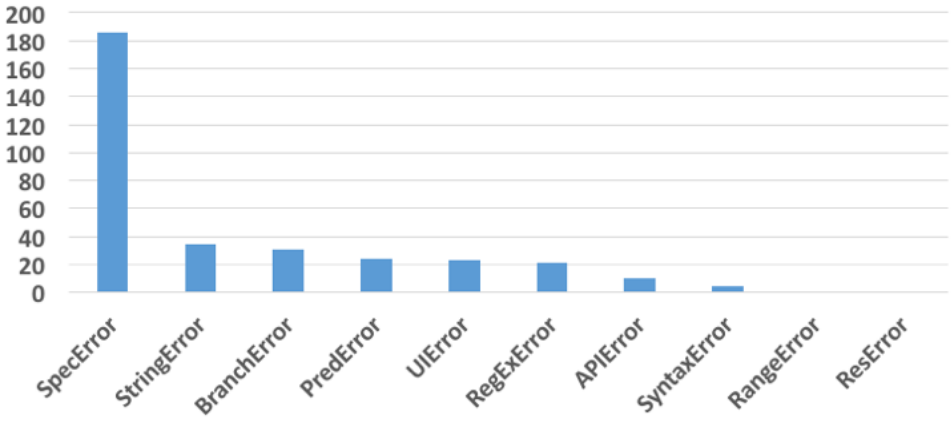

Figure 1: The histogram of unpreventable public bugs under both Flow and TypeScript according to the unpreventability taxonomy.

preventable under TypeScript 2.0 but not under TypeScript 1.8. This result decisively and quantitatively demonstrates the value of TypeScript 2.0's strict null checking.

Comparing Flow and TypeScript Though sharing a similar annotation syntax, Flow and TypeScript differ in terms of expressivity and type variance. These dimensions are hard to quantify. Thus, we compare Flow and TypeScript in terms of their ability to potentially prevent public bugs had they been used when those bugs were introduced and the costs of the requisite annotations.

Flow and TypeScript both catch a nontrivial portion of public bugs. In our dataset, the bugs they can prevent largely overlap, with 8 exceptions: 5 bugs are only Flow-preventable and 3 only TypeScript-preventable.

One bug is only Flow-preventable because Flow has a better support for the `require()` function, Node.js's module importing mechanism, and manages to detect that a module specified as the argument of `require()` does not exist. Another difference arises because Flow has better support for JavaScript's native functions, here `parseInt()`. The remaining three Flow-preventable bugs share a common feature that reveals a weakness in TypeScript's recently introduced handling of **`null`**: TypeScript does not regard concatenating a possibly **`undefined`** or **`null`** value to another of type **`string`** as a type error. For example, TypeScript remains silent on the following statement,

```
1  var x = " " + null + " ";
```

whereas Flow reports a type error:

```
1: ' ' + null + ' '
         ^^^^ null. This type cannot be added to
1: ' ' + null + ' '
   ^^^^^^^^^^ string
```

Without knowing whether TypeScript intentionally allows this behaviour, we cannot judge this decision, but its cost is substantial: TypeScript could have detected 3

more bugs, which amounts to around 5%.

Two of the three bugs that only TypeScript-preventable arise due to Flow's incomplete support for a popular JavaScript idiom, using a string literal as an index. For example, TypeScript succeeds in detecting the reported error in `conanbatt/OpenKaya:45` when annotating two variables used as indexes `i0` and `i1` with `undefined | string | number`, whereas Flow fails with the same annotation. The remaining bug, `sandeepmistry/node-core-bluetooth:1`, arises because of Flow's permissive handling of the `window` object. Below is its error message:

```
node-core-bluetooth/lib/central-manager-delegate.js:146
  }.bind(mapDelegate(self), mapPeripheral(identifier), error));
                     ^^^^
ReferenceError: self is not defined
```

In JavaScript, `self` generally refers to the global object, `window`. This bug is caused by a operating system upgrade, after which the system no longer recognises `self` and forces the developer to use `$self`. Therefore, the fix simply replaces `self` with `$self`. Both TypeScript and Flow are able to infer that `self` has type `Window`. By reading the issue report and the code, we are able to infer that function `mapDelegate` accepts values of only `string` or `number` type. In TypeScript, we add the following annotation to `mapDelegate`'s definition:

```
function mapDelegate(self:string | number) {
```

Upon type checking, TypeScript signals a type error:

```
central-manager-delegate.ts(146,22): error TS2345: Argument of type 'Window' is
not assignable to parameter of type 'string | number'.
```

Flow, on the other hand, even with the same annotation, does not regard `self` being passed to `mapDelegate` as a type error.

The per-Bug Annotation Tax Everything comes at a price. To enjoy the benefits that a static type system brings, a developer often needs to annotate their program. Directly measuring the effort programmers must expend to annotate their programs for a static type system would requires a large-scale, invasive study of two teams of developers, one using static types and other dynamic types, with all the attendant cost and confounds such a large user study would entail. Thus, we resort to under-approximating the annotation tax with two simple, expedient proxies: *token tax*, the number of tokens in the added type annotations, and *time tax*, the time spent adding annotations.

The token tax rests on the intuition that each token must be selected, so this proxy measures the number of decisions a programmer must make when adding type annotations. For a *ts*-preventable bug, we define the token tax as the number of tokens in the annotation needed to trigger a type error on a line involved in causing the bug. Let Δ be a function that returns the syntactical difference between two code snippets and $||\ ||$ be a function that calculates the number of tokens in a code snippet. Then, the token tax is

$$||\Delta(a(b_i), b_i)|| \tag{1}$$

	Token Tax		Time Tax	
	Mean	**Median**	**Mean**	**Median**
Flow	1.7	2	231.4	133
TypeScript	2.4	2	306.8	262

Table 1: The token tax and time tax measured in seconds for TypeScript- and Flow-preventable bugs.

To report the time to annotate, we recorded how long we spent annotating each buggy version in the commit message, creating an electronic labortary notebook.

These measures of the annotation tax are per-bug and underestimate the annotation effort in time and tokens, because our experiment allows us to travel back in time, knowing exactly which region of code will be touched by a fix in response to a bug report. With this knowledge, we locally annotate the region aimed at this specific bug, ignoring unrelated type errors. The developers who originally committed the buggy code lacked this knowledge and would have had to annotate the entire program. Nonetheless, we claim this under-approximation is still useful because when adding code to a statically typed code base, most of the annotations exist, so this per-bug annotation measure is useful as a floor on the annotation cost of annotating fixes.

Using this measure, we answer the question "What is the per-bug annotation tax in number of tokens of TypeScript and Flow, without considering the definition of shims?", finding that on average Flow requires 1.7 tokens to detect a bug and TypeScript 2.4, shown in Table 1. Two factors contribute to this discrepancy: first, Flow implements stronger type inference, mitigating its reliance on type annotations; second, Flow's syntax for nullable types is more compact. As discussed previously, to denote a variable is nullable in Flow, one simply needs to add a `?` before the type annotation, like `?number`, whereas TypeScript requires the use of union type operator, like `number | null | undefined`. The benefit of type inference in saving type annotations is also shown in the median values.

Table 1 exhibits a sharper difference in time tax between Flow and TypeScript. Thanks to Flow's type inference, in many cases, we do not need to read the bug report and the fix, and devise and add a proper type annotation, which leads to the noticeable difference in annotation time.

Cross Pollination In our experiment and case studies, handling modules was the most time-consuming aspect of annotating buggy versions. Flow has builtin support for popular modules, like Node.js, so when a project used only those modules, Flow worked smoothly. Many projects, however, use unsupported modules. In these cases, we learned to greatly appreciate TypeScript community's DefinitelyTyped project. Flow would benefit from being able to use DefinitelyTyped; TypeScript would benefit from automatically importing popular DefinitelyTyped definitions. Flow would also benefit from supporting the use of string literals as array indices.

TypeScript should borrow more null-handling tactics from Flow, as discussed above, like preventing the + operator from simultaneously taking null and string as operands.