FlowJSNicer: A name and type prediction tool on the base of JSNice and Flow

Authors:

- Gabriel Castillo Cortes
- Ameen Eetemadi
- Jian Xu

1. Abstract

Using an integrated approach, we created a system for predicting program properties. The “JSNice” project has showed how to predict program properties by taking a probabilistic approach. The “Flowtype” project, by Facebook, however uses type inference to both suggest and verify variable types of javascript programs. Our early results show that by integrating both approaches we can achieve higher performance for variable type and return value type prediction compared to individual approaches.

2. Introduction

JavaScript is a widely used but loosely typed language. This brings opportunities as well as challenges. Type inference and prediction is of great importance both for code readability, stability and compile time efficiency. In this project, we studied two approaches a) type prediction and b) type inference. We then show a proof of concept that how using type inference we can substantially increase the performance of a probabilistic type prediction system called JSNice. In the following sections we describe how each method works, our proposal and results.

2.1 JSNice/Nice2Predict

JSNice/Nice2Predict is an ongoing project from the Software Reliability Lab at ETH Zurich. Its main objectives are predicting names of local variables and predicting type annotations for function arguments for JavaScript codes. For achieving both objectives they take a probabilistic approach to infer the program properties. They transform the input to a representation that allows to predict either in a syntactical or semantic way the possible value for the properties of interest. In order to suggest either a value or a type, they have a learning algorithm that evaluates available code in large repositories, such as GitHub, and identifies the common traits of similar code in order to suggest the best option. They process the input in multiple stages until they create a suggested output that is readable and correct.

JSNice was the prototype of this approach, and it performs a fair job on both primary objectives. Nevertheless the current state of the project has shifted to first tackle the variable suggestion problem, still using their learning technique. They have renamed the project in 2 sub-projects

Nice2Predict and Unuglify. The first one is the learning part and the second takes care of the processing and output of any given program.

For our project we are proposing the interaction between 2 systems, so in order to allow this we will be using JSNice rather than the later version Unuglify. This will provide a clear overlap of domains that will help us prove the validity of our approach.

2.2 Flow

Flow is a JavaScript static type checker with gradually typed extensions. By employing data-flow and control-flow analysis, it uses a subtyping-based inference algorithm to abstract type information, in order to find type errors and bugs. Using type inference, programs with missing type annotations can benefit from various performance optimizations.

3. Approach

We are taking an integrated approach to enhance the prediction performance of JSNice using the type inference approach taken by Flow. For cases where the type cannot be inferred, we perform prediction using JSNice. We further use the predicted types from JSNice and perform type inference for cases where type could not be predicted.We repeat this process until it converges.

Basically, our tool consists with two adapters adjusting code formats to make the predicted code compatible as an input into both Flow and JSNice. As is shown in Figure 1, a cycle is built between Flow and JSNice, and the type prediction iterations continue until the predicted results from Flow and JSNice converge. As a result, the final code has more type annotations than the codes from Flow and JSNice separately.

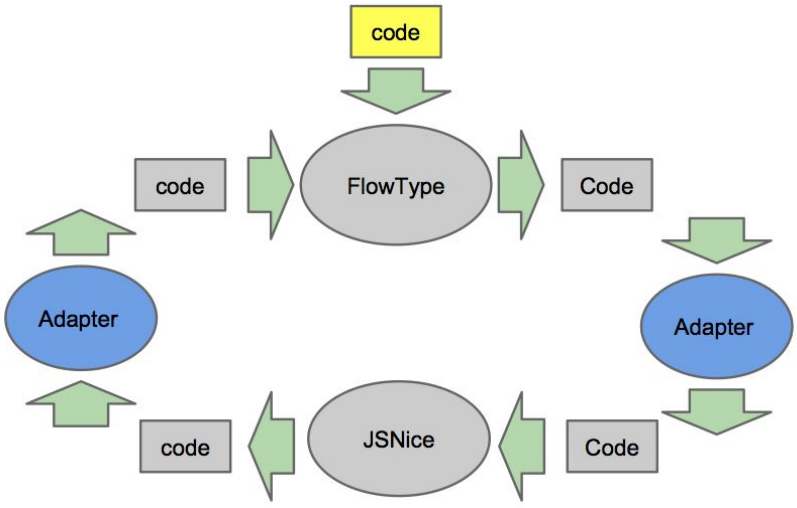

Figure 1. Implementation cycle built between Flow and JSNice using two adapters

4. Implementation and results

Figure 1, shows how our software system is set-up. It is important to note that, for the “JSNice” part, we connect to the restful web service in “http://jsnice.org”. We wrote two different adaptors to be able to make FlowType and JSNice work together. Although we didn’t get a chance to test using the benchmarks yet, we tested number of examples which clearly show the benefit of our approach. Here is one example:

Example

Input:

```
function chunkData(e, t) {
        var n = [];
        var r = e.length;
        var i = 0;
        for (; i < r; i += t) {
                if (i + t < r) {
                        n.push(e.substring(i, i + t));
                } else {
                        n.push(e.substring(i, r));
                }
        }
        return n;
}
```

Flow Only:

```
function chunkData(e, t) {
        var n = [];
        var r = e.length;
/** @type {number} */
        var i = 0;
        for (; i < r; i += t) {
                if (i + t < r) {
                        n.push(e.substring(i, i + t));
                } else {
                        n.push(e.substring(i, r));
                }
        }
        return n;
```

JSNice Only:

```
/**
 * @param {string} str
 * @param {number} step
 * @return {?}
 */
function chunkData(str, step) {
  /** @type {Array} */
  var colNames = [];
  var len = str.length;
  /** @type {number} */
  var i = 0;
  for (;i < len;i += step) {
    if (i + step < len) {
      colNames.push(str.substring(i, i + step));
    } else {
      colNames.push(str.substring(i, len));
    }
  }
  return colNames;
}
```

Combined approach

```
/**
*/
function chunkData( str: string, step: number ) :
Array {
 var colNames : Array = [];
 var len  = str.length;
 var i : number = 0;
 for (;i < len;i += step) {
   if (i + step < len) {
     colNames.push(str.substring(i, i + step));
   } else {
     colNames.push(str.substring(i, len));
   }
 }
 return colNames;
}
;
```

Observations:

1) “Example 1” shows a case where the type suggestion of “flow” is a subset of types predicted by JSNice and therefore, our combined approach does not provide better results than JSNice.
2) In another example (would be available upon request/after submission), we have observed that the types predicted by JSNice has been a subset of the ones predicted by “flow”. In such case, our combined approach would not proved better results than in “flow”
3) Since we do not know in advance, which method would provide better results, our combined approach at a minimum ensures that the user receives the In both #1 and #2, our combined approach achieves the best prediction.

We still need to run the results on a large dataset in order to fully understand the power of our approach.

5. Related work

JSNice is based on the dependency graph of program elements (e.g. variables, functions and operators). Each dependency graph is built for prediction of a particular property. For this study, we are most interested the “type” property of program elements. In our assumption, flow is based on the solving the constraint graph of inflow type data among different type variables.

5.1 Type inference principle and performance of Flow

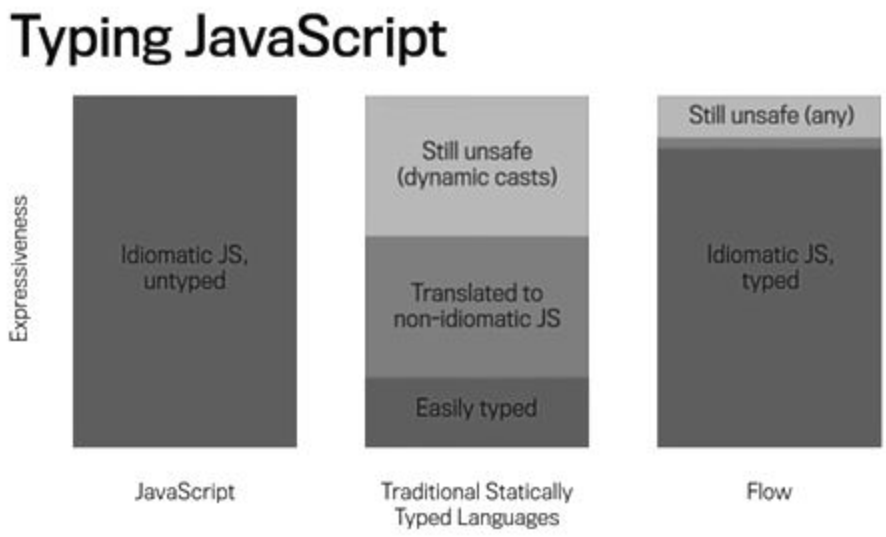

Figure 2. The “Expressiveness” of JavaScript after typed by Flow. Flow suggests safe types without the loss of fluidity. [4]

Flow uses a gradually type system, which checks, at compile-time, for type errors in some parts of a program, but not others, as directed by which parts of the program have been annotated with types. It gradually takes advantage of both static and dynamic type checks. As shown in Figure 2, static check can involve translation, which can lead to safe but ugly code with some loss of fluency, while dynamic cast is still unsafe. However when using gradual type checking which gradually adds explicit type annotations, program can have improved performance and fewer run-time errors. Besides JavaScript, there are gradual type checkers for other programming languages are: Mypy (Python), Hack (PHP), 3.0 (Ruby), etc.
As the author Avik Chaudhuri mentioned, the implementation of Flow is heavily influenced by Pottier's work on subtyping and type inference [5]. In another paper [1], a type inference algorithm co-designed by Chaudhuri is introduced. We assume Flow is in principle similar with that system, of which the basic procedures are:

(1) First, programs are annotated with types, which are simple types like numbers are strings. These annotations are based on the binary operations and other simple type relations.
(2) Then, coercions are applied in the basic types. For coercion, a simple example is: for x=y+z, if y and z are both evaluated to the type *Number* and they are the only two inflows into x, then x can be defined with the type *Number*. The types of most variables can be suggested when the basic coercions of annotated variables are done.
(3) There can be still some types left with unknown relations. Usually, they are in the relation of subtypes, which can be described using subtype constraints. Constraint is an inequation showing one type is a subtype of the other. Thus data flow analysis can be done to compute the conjunction of constraints. A constraint graph can be built to solve the constraints.
(4) A constraint graph is an adapted structure to represent conjunctions. A program should be considered as correct if and only if the constraint graph has a solution. *Closure* is defined as a property which is sufficient to guarantee there is an existence of the solution.
(5) Flow rules should be created for the computation of closure. Different type inference systems have different flow rules, with different efficiency. In Pottier's paper, such type inference rules include: (i) Rule $\mathbf{Var}_l$: for expression x, give a hypothesis that type α, for other variables other than x are not used, they receive type $\top$. (ii) Rule $\mathbf{App}_l$: The expression is with two pieces of code. To have the type of the whole expression, the greatest lower bound of their contexts are computed and their constraint graph are unioned. etc. In the paper of Actionscript [1], another set of rules are defined. These rules iterations can be applied until a fixpoint is reached, when no new flow judgements are derived.

Basically, type inference is done when the solution of flows are computed. It started from annotating the known types. This method is similar to the dependency paragraph method in JSNice, since they all need the known types to find out the unknowns. However, Flow seems more powerful in type annotations. It is probably because the iterations of the subtyping constraints to find out the closure is more efficient in digging out all the possible relations between different types.For better accuracy, techniques like blame recovery can be added to adjust the prediction result.

5.2 The type inference performance of JSNice

JSNice [2] uses two concepts to describe the prediction effect: “Precision” and “Recall”. “Recall” is the percentage of predictions that JSNice makes other than “?”, while “Precision” is the percentage that JSNice making the right prediction, using manually provided JSDoc annotation as a standard of reference. The “Recall” will not be high when doing type prediction of JavaScript applications with complex relationships. The system in the paper achieves 66.9% “Recall” when doing one such prediction. It is reasonable that JSNice does not infer all the types, because some types are not safe to infer, and even for the types that are safe to infer, they may be from very complex relations and trying to type infer these variables can be very complex.

The paper also evaluates the effect of structure when making predictions, this is done by disabling relations between unknown properties and predictions on that network. The result shows that dropping the structure will slightly increase the “Recall” but greatly decrease “Precision”. Thus analyzing the unknown properties jointly will definitely help to predict the result.

5.3 Why Flow has a better (if so) performance in type prediction

JSNice is a system based on data training. However, unlike doing name prediction, adding more training data may not help to significantly increase performance of type prediction. The training data is from the JavaScript program having been annotated with types. There can be many subsets of type specific to only one particular program. And these type relations can be irrelevant for type prediction for new programs. Thus, irrelevant type relation is a restriction in the type training data.

Instead, adding more (semantic) relationships is of higher importance. Flow may be a type system with more good-defined (semantic) relationships. A good design of inference rules of compilation and computation of closure will increase prediction precision as well as the algorithm efficiency.

5.4 The types that both Flow and JSNice cannot infer

There are some types that both Flow and JSNice fail to infer, one important reason is that these types are not safe to infer. These cases include [1]:

(1) Local function that escapes: When they are available to be called in the wild, they are not safe to infer because not every call from wild to the function is available for analysis.

(2) Unions of Function types to higher-order may not work, since unlike the union of base types, union can be very complicated and just directly doing that may resulting in errors.

(3) Function callable by existing code, because analysis cannot be guaranteed to include every call of such functions.

(4) The parameter types of the functions that are callable by existing code is not safe to infer. If such types inferring into other variables, then those variables are also not safe to infer.

6. Conclusions

This project uses both JSNice and Flow, and integrates them in a single tool to do a better type annotation job. Besides doing type annotation, we also want to do some name prediction for obfuscated code and make it readable. For this task, the program property to predict could be variable name or variable type (for the dynamically typed languages). Another reason for type annotation is to do an efficient type check, since JavaScript is a programming language in nature of dynamically typing, which binds the language performance. A method of gradual type inference can be used to enhance the type prediction and check performance.By using two adapters, a iteration loop is built between JSNice and Flow. In the limited test examples, Flow does better type prediction in one while JSNice does better in the other. The case that our tool does better than both of them is yet to be found. In the future, we are going to test our tool with larger dataset base, for the sake of potential improvement of Flow and JSNice.

7. Reference

[1] The Ins and Outs of Gradual Type Inference A. Rastogi, A. Chaudhuri, B. Hosmer, in *ACM SIGPLAN Notices* (ACM, 2012; http://dl.acm.org/citation.cfm?id=2103714), vol. 47, pp. 481–494.

[2] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '15). ACM, New York, NY, USA, 111-124.

[3] Pottier, F. Type Inference in the Presence of Subtyping: from Theory to Practice. (1998). at <https://hal.inria.fr/inria-00073205/document>

[4] https://www.youtube.com/watch?v=M8x0bc81smU&t=768

[5] https://news.ycombinator.com/item?id=8625222