Computer Science Project Proposal

Building a Gradual Type System for JavaScript

Christopher J.O. Little, Emmanuel College

October 24, 2014

Project Supervisor: Dr Kathryn E. Gray

Director of Studies: Dr Jonathan Hayman

Project Overseers: Dr Markus Kuhn & Dr Neal Lathia

1 Introduction

Static type systems can offer a number of advantages over their dynamic counterparts, including earlier detection of certain programming errors, better development tools and potential for compiler optimisations. Proponents of dynamically typed languages may argue that static type systems are too inflexible and interfere with the natural thought processes of the developer.

JavaScript is a dynamic language with little to no built-in support for types. A few projects have attempted to integrate optional explicit type annotations to dialects of ECMAScript (JScript.NET, ActionScript 3.0, ECMAScript 4 and more recently TypeScript), but the requirement for explicit type annotations limits the utility of these for existing JavaScript projects. Often this requirement is relaxed such that type annotations are optional, but in these cases it is not evident to the developer which parts of the codebase are type-safe, and which have not been checked.

Both of these problems could be resolved with a gradual type inferring system. Gradual typing is a modern technique allowing safe interoperability between typed and untyped portions of a single program. The technique allows us to guarantee that only dynamic properties of the code are able to cause runtime errors, retaining the utility of statically type-checking part of the code. [1, 2]

Using type inference means no extensions to the language are required, such that the system could be used to retrospectively analyse existing code. Due to the highly dynamic nature of JavaScript, it will not always be possible to statically determine the type of an expression. This is where the gradual typing nature of the system becomes critical - we can safely assert that not only are

the statically analysed portions of the program free of type errors, they are also guaranteed to remain so even after interaction with potentially untyped code.

2 Starting Point

I am comfortable writing programs in JavaScript and familiar with many of the semantic quirks of the language. Many of the concepts required for the execution of this project were introduced in the Part IB courses Compiler Construction and Semantics. The Part II courses on Optimising Compilers and Types will also be useful.

I have already done some preparatory reading in this area, on types in general [5], on the design and semantics of JavaScript [3, 4], and specifically on gradual typing [6, 7] and on TypeScript [8], a recent project by Microsoft extending the JavaScript language with type annotations.

For the implementation of the project, it will be necessary to construct an abstract syntax tree from JavaScript source, and, after modification of the tree, reconstruct the JavaScript source code. Since the parsing and reconstruction of JavaScript is not of particular relevance to the core focus of this project, and in order to integrate with existing solutions in the world of JavaScript, I will use an existing system for this - UglifyJS.

UglifyJS is a tool for minification and compression of JavaScript code, itself written in JavaScript. It uses a recursive descent parser to construct an abstract syntax tree from JavaScript source code, then modifies the tree (renaming identifiers and similar) before reconstructing minimal source code.

3 Resources

For this project I will mainly use my own machines - either a desktop (Ubuntu 14.04, 3.6GHz i7, 8GB RAM, 2x 1TB HDD) or laptop (Windows 8.1, 1.8GHz i7, 8GB RAM, 1TB HDD) as convenient. I will use Git for version control, with remotes set up on Github and on a VPS hosted by BHost for backup purposes. I require no additional special resources.

4 Work to be Done

The project can be broken into the following components:

1. Definition of a formal type system along with an operational semantics for the subset of JavaScript which is supported. The system will have the following properties to prove:

   Progress: If an expression is well-typed, then either it can reduce further to a new expression, or it is a value (and cannot reduce further).

Type Preservation: If an expression e is well-typed and reduces in some number of steps to another expression, then that expression is also well-typed.

Decidability of Type Checking: It is decidable for any expression in the language whether or not it has a given type.

Type inference: It is possible to find a type for any expression in the language, or to show that no such type exists.

In addition, it will be necessary to prove that any runtime errors which do arise have been caused by untyped code.

2. Implementation of a static type checker for the above type system. This will need to include a type inference algorithm in order to make any deductions about the types of JavaScript objects. This algorithm could be based upon a variation of the Hindley-Milner algorithm.

3. Implementation of a source-to-source compiler to implement gradual typing, allowing code which has had its type checked by the above type checker to safely interface with unchecked code. The compiler will inject two kinds of wrappers around objects to achieve this. Guard wrappers protect a well-typed object from untyped code by performing dynamic type checks on parameters passed to its methods, or creating a mimic wrapper for the parameter. It will also create guard wrappers for return values (which are well-typed). Mimic wrappers protect well-typed code from an untyped object by checking for the existence of the object's properties and methods when these are requested, and checking return types or creating a mimic wrapper around the return value when this is not possible. They will also create guard wrappers around parameters (which are well-typed and entering untyped code). The operation of this compiler will be validated by running it on a series of test cases, and also on some existing production code. These test cases will be written alongside the type checker.

5 Success Criteria

The project will be considered a success if the following points have been achieved:

1. The design of an operational semantics and type system for a subset of JavaScript including:
   (a) Primitives, constants and object literals
   (b) The following control flow statements
      i. seq (... ; ...)
      ii. block ({ ... })
      iii. if ... else

iv. while
v. for
vi. continue
vii. break

(c) Assignment via =, and all shorthand assignment operators (such as +=)

(d) Arithmetic operators +, -, *, /

(e) Logical operators !, &&, ||

(f) String concatenation with +

(g) Function definition

2. An implementation of a type checker for a subset of JavaScript defined by the list above.

3. An implementation of a source-to-source compiler injecting code to provide type-safe wrappers around dynamic type-unsafe objects.

Evaluation of the design of my type system will be achieved through a formal proof of soundness. Evaluation of the implementation will be through testing to monitor performance and memory overhead.

6 Possible Extensions

Arrays in JavaScript pose a difficult problem, since the types of their contents are unrestricted. My project will thus be limited to programs which do not include arrays, but it would in principle be possible to extend the type system to handle them. This extension would also allow handling of variadic functions, which access their actual parameters through the `arguments` array.

A common idiom in JavaScript is to initialise an object with certain properties and methods, but then to modify the object later on, either adding or removing properties. This kind of behaviour is difficult to support in a static type system, but could be supported with gradual typing.

Objects in JavaScript can be created by a literal object initialiser, or in a more object-oriented style with the `new` keyword. Using `new` with a constructor creates a new object following an inheritance chain and binds the special `this` variable to the new object. The inheritance involved in this method complicates typing, but this would be a valuable extension to my project, as `new` is a common way to initialise objects.

It would also be informative to run the project on larger JavaScript programs, in order to measure the impact of the compiler on performance and memory in the presence of a larger codebase. An extension might thus be to run the compiler on at least a portion of Mozilla's SunSpider JavaScript benchmark series.

7 Timetable

Weeks 1 & 2 (24^{th} Oct – 6^{th} Nov)

Continue learning about techniques used in gradual typing, and implementations of Hindley-Milner. Prepare an initial design for the type system such that most of the core features of my subset of JavaScript have a defined operational semantics and type judgement.

Weeks 3 & 4 (7^{th} Nov – 20^{th} Nov)

Create a trivial implementation of the type checker which simply rejects everything, and a trivial compiler which generates an AST and re-generates source code from it. Complete type system if required.

Weeks 5 & 6 (21^{st} Nov – 4^{th} Dec)

Extend the type checker and compiler allowing primitives and constants. Prove safety and write test cases for these new cases. Begin extension for variable initialisation and assignment.

Weeks 7 & 8 (5^{th} Dec – 18^{th} Dec)

Complete extension for assignment, and add extension for object literals. Prove safety and write test cases for this extension.

Weeks 9 & 10 (19^{th} Dec – 1^{st} Jan)

Extend to support all control flow statements. Prove safety and write test cases for this extension.

Weeks 11 & 12 (2^{nd} Jan – 15^{th} Jan)

Ensure development is on schedule, and prepare progress report.

Weeks 13 & 14 (16^{th} Jan – 29^{th} Jan)

Extend the type checker and compiler for function definitions and arithmetic and logical operators. Prove safety and write test cases for this extension. By this point, all core features of the project should be implemented.

Weeks 15 & 16 (30^{th} Jan – 12^{th} Feb)

Finalise development and fix any bugs. Complete test harness.

Weeks 17 & 18 (13^{th} Feb – 26^{th} Feb)

Evaluate performance and additional memory overhead of compiled JavaScript. Complete the proofs of soundness if not yet done.

Weeks 19 – 23 (27^{th} Feb – 26^{th} Mar)

Write dissertation and refine after feedback before submission. Prepare plan for tackling any extension work.

Weeks 24 – 28 (27^{th} Mar – 23^{rd} Apr)

Extend project as far as possible, and update dissertation as appropriate.

References

[1] S. Tobin-Hochstadt and M. Felleisen: *Interlanguage Migration: From Scripts to Programs.* In: Proc. of ACM Dynamic Languages Symposium. (2006)

[2] P. Wadler and R.B. Findler: *Well-typed programs can't be blamed* In: Proc. ACM Workshop on Scheme and Functional Programming (2007)

[3] A. Guha, C. Saftoiu, and S. Krishnamurthi: *The Essence of JavaScript* In: ECOOP (2010)

[4] G. Richards, S. Lebresne, B. Burg and J. Vitek: *An Analysis of the Dynamic Behavior of JavaScript Programs* In: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (2010)

[5] B. Pierce: *Types and Programming Languages* (2002)

[6] K. Gray: *Safe Cross-Language Inheritance* In: ECOOP (2008)

[7] E. Meijer and P. Drayton *Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages* In: OOPSLA Workshop on Revival of Dynamic Languages (2004)

[8] G. Bierman, M. Abadi and M. Torgersen: *Understanding TypeScript* In: ECOOP (2014)