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 builtin 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. Since no extensions to the language are required, such a system could be used retrospectively to 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 type-unsafe 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. I have also done some reading in this area, in particular Benjamin Pierce's book on Types and Programming Languages, and a paper on the the dynamic behaviour of JavaScript programs by Richards, Lebresne, Burg and Vitek.

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 JavaScript source again. Since the parsing and reconstruction of JavaScript is not of particular relevance to the core focus of this project, I will use an existing system for this - UglifyJS2.

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 the subset of JavaScript which is supported. The system should have the following properties:

   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 e', then e' 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.

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.

5 Success Criterion

The project will be considered a success if I have produced a type system for which I have proven Progress and Type Preservation for a particular subset of JavaScript including primitives, object literals and functions including all control flow operators and most assignment, comparison and arithmetic operators. In addition, a type checker will be implemented according to this type system, as well as a source-to-source compiler allowing gradual typing while preserving semantics of the original program and minimising impact on performance.

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.

7 Timetable

tbc

< Superficially, it seems sensible to an empty type system (i.e. nothing can be checked), along with a trivial type checker (always returns false) and compiler (constructs AST then deconstructs it again). I can then iteratively add typable expressions to my system, and update the type checker and compiler appropriately. Following this, the first non-trivial expression added would likely represent the most work, but subsequent expressions would likely use a lot of the same scaffolding and take less time to implement. I feel like this might work more effectively than the alternative of following the 3 steps outlined in "Work to be done" sequentially. >