Shrinking Desugaring

A Master's Project Report

Junsong Li

Advisor: Shriram Krishnamurthi

Introduction

Desugaring is a powerful tool for studying programming language semantics. Various systems, such as λ_{s5} [3] and λ_{π} [4], use desugaring to explain the semantics of real-world programming languages. The essence of desugaring is to reduce language complexity by translating the source language to a small, well-designed core language. Desugaring provides a new way to make tools for the surface language; tools only need to understand the small core language in order to process the programs in the surface language. However, because the core language is much more concise than the source language, the desugaring process will inevitably produce a large code size that presents a huge challenge on building sophisticated tools.

In this project, we propose shrinking the desugared code by two types of transformation techniques. The first uses conventional semantics-preserving transformations inspired by compiler optimization techniques. The other, more interesting and unconventional technique uses semantics-altering transformations which intentionally alter the meaning of the program. The main focus of this project is the semantics-altering technique.

Background

λ_{s5} is a core language for the strict mode of the ECMAScript 5 (ES5) language [2], commonly referred to as JavaScript. ES5 has a test suite, *test262* [1], which comprehensively covers language features. Because the test suite is a critical component of this project for verifying whether a semantics-preserving transformation is implemented correctly, this project uses λ_{s5} as a concrete language for applying the idea of shrinking desugaring.

Take desugaring an identity function as an example of code bloat:

```
function id (x) {
    return x;
}
```

Desugaring in λ_{s5} follows the ES5 spec to generate code, including introducing the prototype for the function object, setting up properties of the object, setting up the scope for resolving the local naming issues, etc. The simple identity function is desugared to a massive λ_{s5} program that contains 119 AST nodes, in which we find that code bloat comes from two sources: the inherent complexity of the source language, which is the main source of code bloat, and the context-insensitive, recursive-descent desugaring process, which generates obvious redundant code.

A Summary of Transformations

We propose two types of transformations to shrink the code: semantics-altering transformations to reduce the complexity of the source language by weakening language features, and semantics-preserving transformations to clean the redundant code produced by the desugaring process.

Semantics-altering Transformations

Semantics-altering transformations shrink the code size by purposely altering the semantics. They are implemented based on assumptions about language use.

Fixing Function Arity. JavaScript supports variable-arity functions. Variable arity can be practically used to support optional arguments, but it can be error-prone as well, since arity-mismatch errors are not reported to programmers. This transformation simply disables the variable arity features.

Function Restoration. Since functions are objects in JavaScript, a function cannot be directly transformed to lambda form in λ_{s5}. However, if a function is not used as an object, it is reasonable to only maintain the function in a simple lambda form. This transformation uses a good heuristic to restore a function object to lambda: function objects that do not use `this` can be restored to λ_{s5} functions.

Simplifying Arithmetic Operations. Arithmetic operations have a complicated semantics in JavaScript. This transformation simplifies the semantics of these operations. For example, the new semantics only supports "+" working on the arguments of the same type, either strings or numbers, but not both at once.

Identifier Restoration. JavaScript uses a *context object* to handle scope, and thus a reference to an identifier actually becomes a field lookup operation in the desugared code. This transformation directly maps these field lookup to identifiers of λ_{s5}. It produces a simpler semantics that can be exploited by other transformations.

Unsafe Assertion Elimination. The JavaScript spec is littered with implicit type conversions and type checks. λ_{s5} manifests these conversions, and generates a lot of code. This transformation removes them under the assumptions that code will never fail these checks.

Semantics-preserving Transformations

To justify the shrinkage achieved by semantics-altering transformations, we must also implement semantics-preserving ones to show that the same shrinkage could not be achieved by simply implementing language-specific, conventional semantics-preserving transformations. Thus we also implement the following transformations:

Assignment Conversion. This transformation substitutes `let` bindings for assignments, introducing a simpler semantics that other transformations can exploit.

Constant Propagation. This transformation propagates let-bound constant values.

Constant Folding. This transformation evaluates constant expressions.

Dead Code Elimination. This transformation deletes unreachable code, useless let bindings, and statements that have no side effect.

Non-constant Propagation. Desugaring often introduces `let` bindings to avoid duplicate evaluation, though they are not always necessary. This transformation propagates and eliminates these bindings. It also propagates single-use functions.

Assertion Elimination. This is a variant of *Unsafe Assertion Elimination*. It only removes assertions that are *demonstrably* redundant.

Function Inlining. This transformation inlines functions that are propagated by *Constant Propagation* and *Non-constant Propagation*.

Environment Cleaning. λ_{s5} ships with an *environment* that implements built-in JavaScript functions and objects. Not all of these built-in functions and objects are always used in the program. This transformation removes unused functions and objects in the *environment* using extra information that *Dead Code Elimination* does not have.

Evaluation

We evaluate the effectiveness of these transformations by testing them against *test262* and a collection of libraries in the wild.

For semantics-preserving transformations, we test the correctness against *test262* to ensure they do not accidentally alter the semantics. We also measure the shrinkage of *test262* (measured per test) and the library collection (measured per library) for these transformations. Testing shows that

1. These transformations produce code that continues to pass the test suite and thus they preserve the semantics.
2. The shrinkage of user code is around 6%, which means that the desugaring process is not the main source for the code bloat of λ_{s5} programs.

For semantics-altering transformations, it is meaningless to only test the correctness due to the nature of these transformations. Instead, we focus on the trade-off between code size and the correctness to show whether these transformations still generate practically useful code (measured by correctness) and whether they shrink more code (measured directly by shrinkage). Testing shows that

1. Each individual semantics-altering transformation is not *too* aggressive, the correctness ranging from 80% to 100%.
2. The more transformations we apply, the more code we can shrink while trading the correctness. The shrinkage can be on the order of 50%.

When both semantics-altering and semantics-preserving transformations run, testing shows that the combined effect is very close to the sum of them individually. Combining these results, we conclude that most code bloat in λ_{s5} is semantic bloat.

Conclusion

The code bloat problem is actually a general problem, not specific to λ_{s5} only. λ_π, a tested Python semantics, also shares this problem. The same identity function written in Python is desugared to over 100 AST nodes in λ_π. The heart of this project is a case study to explore semantics-altering transformations. We demonstrate the effectiveness of these transformations. We also demonstrate how assumptions of language use can confine code bloat.

Acknowledgments

This project was done in collaboration with Justin Pombrio, whose great insight and dedication made this project and its related paper possible. I am also thankful for Joe Politz, who provided many insightful ideas. Finally, I would like to thank my advisor, Professor Shriram Krishnamurthi, for all his help and his patience on this work and for his valuable guidance over the past two years.

Bibliography

[1] ECMA International. ECMAScript Language test262. `http://test262.ecmascript.org/`.

[2] ECMA International. ECMAScript Edition 5.1. 2011. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`.

[3] Joe Gibbs Politz, Matt Carroll, Benjamin S. Lerner, Justin Pombrio and Shriram Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In *Dynamic Languages Symposium*. 2012.

[4] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu and Shriram Krishnamurthi. Python: the full monty: A tested semantics for the Python programming language. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 217–232. New York, NY, USA, 2013. ACM.