JavaScript Module System: Exploring the Design Space

Junhee Cho Sukyoung Ryu
KAIST
{ssaljalu, sryu.cs}@kaist.ac.kr

Abstract

While JavaScript is one of the most widely used programming languages not only for web applications but also for large projects, it does not provide a language-level module system. JavaScript developers have used *the module pattern* to avoid name conflicts by themselves, but the prevalent uses of multiple libraries and even multiple versions of a single library in one application complicate maintenance of namespace. The next release of the JavaScript language specification will support a module system, but the module proposal in prose does not clearly describe its semantics. Several tools attempt to support the new features in the next release of JavaScript by translating them into the current JavaScript, but their module semantics do not faithfully implement the proposal.

In this paper, we identify some of the design issues in the JavaScript module system. We describe ambiguous or undefined semantics of the module system with concrete examples, show how the existing tools support them in a crude way, and discuss reasonable choices for the design issues. We specify the formal semantics of the module system, which provides unambiguous description of the design choices, and we provide its implementation as a source-to-source transformation from JavaScript with modules to the plain JavaScript that the current JavaScript engines can evaluate.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords JavaScript, module system, source-to-source transformation

1. Introduction

JavaScript [4] was originally developed as a simple scripting language but now it is one of the most widely used programming languages. The main advantage of using JavaScript in web documents is to support dynamic interaction with users. JavaScript enables dynamic partial updates of web documents by communicating with servers through HTTP and updating web documents via browser APIs for Document Object Model (DOM) [25]. Thanks to the capability, 98 out of the 100 most visited websites according to Alexa [11] use JavaScript [8], and its use outside client-side web programming keeps growing. Large stand-alone projects such as node.js [12] for building scalable network applications use JavaScript and web applications on various platforms including Samsung Smart TV SDK [21] and Tizen SDK [5].

However, JavaScript does not provide any language-level module system. Simple scripts may not need to maintain namespace but a module system is necessary for programming in the large. Because JavaScript does not have the standard libraries, JavaScript developers use more than a dozen libraries: jQuery [13] focuses on improving the interaction between JavaScript and HTML, MooTools [24] emphasizes animation supports, Prototype [23] focuses on adding new features to JavaScript, and other libraries have their own strength. Due to the lack of the single standard libraries, JavaScript developers often use multiple sets of third-party libraries with duplicated functionalities in one project to selectively use functionalities from them. Similarly, some JavaScript projects use even multiple versions of a single library to use both new API functions introduced in a newer version and old API functions deprecated in the newer version. While multiple versions of a single library obviously share symbols with the same names, different libraries often use the same symbol names too, which lead to name conflicts when a single project uses them. For example, because both jQuery and Prototype use the symbol `$`, if an HTML document includes both libraries by `<script>` tags, the library included later overrides the symbol `$` from the other library with its own `$`. Figure 1 shows an example from the jQuery API document [14], which uses multiple versions of a single library and resolves name conflicts explicitly; it loads two versions of jQuery, `jquery-1.9.1.js` and `jquery-1.6.2.js`, and then re-

```
<!DOCTYPE html><html>
<head>
  <script src="http://code.jquery.com/jquery-1.9.1.js">
  </script>
</head>
<body>
<div id="log"><h3>Before $.noConflict(true)</h3></div>
<script src="http://code.jquery.com/jquery-1.6.2.js">
</script>
<script>
var $log = $( "#log" );
$log.append( "2nd loaded jQuery version ($): " +
             $.fn.jquery + "<br>" );
/* Restore globally scoped jQuery variables to
   the first version loaded (the newer version) */
jq162 = jQuery.noConflict(true);
$log.append( "<h3>After $.noConflict(true)</h3>" );
$log.append( "1st loaded jQuery version ($): " +
             $.fn.jquery + "<br>" );
$log.append( "2nd loaded jQuery version (jq162): " +
             jq162.fn.jquery + "<br>" );
</script></body></html>
```

Figure 1. Multiple versions of jQuery in one HTML document, an excerpt from the jQuery API document [14]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MODULARITY '14, April 22–26, 2014, Lugano, Switzerland.
Copyright © 2014 ACM 978-1-4503-2772-5/14/04. . . $15.00.
http://dx.doi.org/10.1145/2577080.2577088

```
var module = function () {
  var privateVar = "accessible only from within module";
  function privateMethod() {
    alert("accessible only from within module");
  }
  return {
    publicProp: "accessible as module.publicProp",
    publicMethod: function() {
      alert(privateVar);
      privateMethod();
      alert("accessible as module.publicMethod");
    };
  };
}();
```

Figure 2. JavaScript module pattern

stores jQuery's globally scoped variables to the first loaded jQuery, `jquery-1.9.1.js`, by `jQuery.noConflict(true)`.

To alleviate the problem, the JavaScript community has used the "module pattern" [19]. The module pattern is to use functions to simulate modules by assigning an anonymous function to a namespace object and adding "private" members to the function. For example, Figure 2 shows a simple use of the module pattern where the anonymous function returns an object with public members which may use private members defined in the function. The module pattern provides some forms of namespace manipulation but such a programming pattern is verbose and error prone, and programmers are responsible for checking that they follow the pattern correctly.

Given the growing popularity of JavaScript as a general purpose language for programming in the large and the need for a standard, language-level support for namespace manipulation, the next release of JavaScript (JavaScript.next) plans to include a module system. The ECMAScript Harmony proposals [2] are a collection of proposals for the next ECMA-262 language specification, and most of the proposals have already been incorporated into the ECMAScript 6 specification draft [3]. While the Harmony proposals include a module proposal [10], the proposal description is very brief and informal. Even in the ECMAScript 6 specification draft, the sections related to modules are empty. The Harmony module proposal describes only the high-level ideas and the design principles of the module system succinctly; it does not specify the complete semantics of modules especially the various interactions with the existing JavaScript language features.

Despite of the current status of the module proposal, the long-awaited module system gets eagerly adopted in various ways. Kang and Ryu [16] present a formalization of the core Harmony module proposal that does not include dynamic module loading. They introduce a formal specification and implementation of the module system by desugaring JavaScript extended with modules to λ_{JS} [9], a core-calculus of JavaScript. The Traceur [6] compiler translates a JavaScript.next program to a JavaScript program, which enables developers to use the new JavaScript.next features in advance. It provides various new features including a module system and asks for user inputs to reflect the feedback in the ECMAScript standards process. Among many variant languages of JavaScript, TypeScript [18] is a typed superset of JavaScript that compiles to plain JavaScript. It supports classes, modules, and interfaces to improve static checking and verification of TypeScript programs.

Unfortunately, the immature status of the Harmony proposal causes discrepancies between the various module extensions of JavaScript. While Kang and Ryu's module system clearly describes the module semantics via desugaring to λ_{JS}, because the semantics of λ_{JS} is very different from that of JavaScript, it is difficult for JavaScript developers to understand the semantics in terms of λ_{JS}. On the contrary, Traceur and TypeScript do not describe their module systems clearly; Traceur partially implements the module proposal and TypeScript provides some variant of the proposal.

In this paper, we investigate the design issues of the JavaScript module system with concrete code examples. While one of the goals of the module system is orthogonality from existing features, the informal proposal reveals some ambiguous semantics of the module system and interactions with existing features. First, naïve source-to-source transformations by Traceur and TypeScript provide inconsistent semantics of import and export statements. For example, when a Traceur module imports an exported variable from another module, the importing module assigns the current value of the variable to the imported variable without any connection to the exported variable, which does not satisfy the module proposal. Second, the proposal specifies the semantics when there are conflicts between import declarations, but it does not specify any semantics when there are conflicts between function, variable, module, and import declarations. Third, while every JavaScript object has its prototype[1], a module instance object that is a first-class object that reflects the exported bindings of an evaluated module does not have any prototype. The unique difference of module instance objects from ordinary JavaScript objects leads to peculiar semantics of module instance objects. Finally, the module proposal does not specify the interactions between module declarations and dynamic code generation functions such as `eval`. We believe that an exploration of the design space will help the ECMAScript committee to design the module system more clearly and reasonably, and the third-party developers to support the module system correctly.

To rigorously discuss and describe the JavaScript module system, we present a formal specification of the module system via source-to-source transformation rules. Because the ECMAScript language specification describes the semantics informally in prose, we developed a formal specification and implementation of a Scalable Analysis Framework for ECMAScript (SAFE) [17] and made it publicly available [15]. We provide a formal specification and an implementation of the module system by extending SAFE with the rewriting rules from JavaScript with modules to the plain JavaScript. By supporting JavaScript.next features today via source-to-source translation precisely, JavaScript application developers can try future features in advance and contribute their feedback to the design of JavaScript.next, and JavaScript engine developers can apply the same technique to their existing engines without modifying the current implementation largely. Our implementation of the module system is also available to the public via the SAFE repository.

The contributions of this paper are as follows:

- We describe design issues of the JavaScript module system with concrete examples, and we discuss reasonable design choices.
- We present a formal specification of the module system, which enables to discuss various design choices formally.
- We provide an implementation of the module system so that JavaScript developers can try future JavaScript features today and JavaScript engine developers can support the features today.

The remainder of this paper is organized thus. In Section 2, we introduce design issues in the JavaScript module system with the comparison of how Traceur and TypeScript support the module system, and we discuss the design issues and reasonable choices. Section 3 describes a formalization of the module system and it addresses some of alternative semantics depending on the design choices. It also proves two safety properties of the module system: validity of module environments and isolation of namespaces. In Section 4, we describe our implementation of the module system

[1] Each JavaScript object has an internal link to its *prototype* as a way of inheritance and a list of prototype links is a *prototype chain*.

```
(function Foo() {
    var foo = "foo", bar = 42;
    (function Bar() {
        bar;            // undefined;
        var bar = "bar";
        bar;            // "bar";
    })();
})();
```

Figure 3. Static variable declaration in a function scope

```
(function Foo() {
    var foo = "foo", bar = 42;
    (function Bar() {
        var bar;        // variable declaration
        bar;            // not yet initialized
        bar = "bar";    // variable initialization
        bar;            // after initialization
    })();
})();
```

Figure 4. Same semantics with Figure 3

```
(function Foo() {
    var foo = "foo", bar = 42;
    (function Bar() {
        bar;            // 42;
        eval("var bar = 'bar';");
        bar;            // "bar";
    })();
})();
```

Figure 5. Dynamic variable declaration in a function scope

and discuss limitations of our approach. Section 5 discusses related work and we conclude in Section 6.

2. Design Issues in the JavaScript Module System

The brief informal description of the Harmony module proposal leads to discrepancies between implementations of the module system. Both Traceur and TypeScript provide inconsistent semantics for import and export declarations, for example. To make the module proposal more elaborate and precise, and to prevent the current module implementations from diverging from the proposal semantics, we discuss four important design issues:

- Unclear semantics of import and export declarations
- Incomplete semantics of name conflict resolution
- Inconsistent semantics of prototypes of module instance objects
- Unclear semantics of the module system in the argument of the `eval` function

We describe each design issue, compare the semantics supported by Traceur and TypeScript, discuss possible semantics that are compliant to the Harmony proposal, and propose a reasonable semantics among the alternatives.

2.1 Import and Export Declarations

Design Issue The Harmony proposal does not describe the semantics of the import and export declarations completely. The proposal specifies their semantics as follows:

```
module Foo {
    export var foo = "foo", bar = 42;
    module Bar {
        export bar;
        bar;                    // 42;
        eval("var bar = 'bar';");
        bar;                    // 42? "bar"?
    }
    export Bar;
}
Foo.Bar.bar;                    // 42? "bar"?
```

Figure 6. Dynamic variable declaration in a module scope

> "Export declarations declare that a top-level declaration in a module is visible externally to the module."
>
> "Import declarations bind another module's exports as local variables."

Because the Harmony proposal does not specify the interactions between the import and export declarations and other existing JavaScript features, there are several cases where the semantics is not clear.

First, the proposal does not provide a clear semantics of the interactions between the evaluation of import and export declarations and that of variable and function declarations. Before describing the interactions, let us review the JavaScript semantics for the evaluation of variable and function declarations. When a JavaScript engine evaluates a program code or a function body code, it first binds all the variable and function declarations in the code and then evaluates the code. For a variable declaration with an initialization expression, the variable declaration is conceptually divided into a variable declaration without any initialization expression and an assignment expression to bind the initialization expression to the variable. Thus, variable and function declarations are bound before actually evaluating any other statements; in a single scope, variables or functions may be used before their definitions textually. For example, Figure 3 shows that two occurrences of `bar` in the body of the function `Bar` evaluate to different values. The first reference to `bar` evaluates to `undefined` because it is already declared but not yet initialized, and the second reference to `bar` evaluates to `"bar"` because it is initialized with the value. In effect, the code in Figure 4 has the same semantics with the one in Figure 3. On the other hand, a variable declaration in a code string passed to the `eval` function as its argument is not evaluated as a variable declaration in the enclosing code; the variable declaration is bound after evaluating the `eval` function call, which behaves differently from static variable declarations. Consider the example in Figure 5. The first reference to `bar` evaluates to `42` instead of `undefined` because the variable `bar` is not yet declared in the body of the function `Bar` because the `eval` function call is not yet evaluated.

Second, similarly for the first case, the proposal does not specify the semantics of the interactions between the import and export declarations and other declarations in a module scope. Because a module scope is analogous to a function scope as the module pattern illustrates, a similar issue arises: what happens when the evaluation of the `eval` function declares a variable that is already declared in the module scope? It is not clear whether a variable declared by the `eval` call shadows the already declared variable with the same name. It is not clear either whether such a dynamic variable declaration should affect the values of the exported variables. For example, consider the example in Figure 6. The second reference to `bar` in the module `Bar` and the property access to `Foo.Bar.bar` outside the module may evaluate to `42` if the dy-

```
module Foo {
    var foo = "foo";
    module Bar {
        export var foo = "bar", bar = "bar";
        eval("delete foo; delete bar;");
    }
    export { foo, Bar };
}
Foo.Bar.foo;         // "bar"? undefined? error?
Foo.Bar.bar;         // "bar"? undefined? error?
```

Figure 7. Export declarations and the `eval` function

```
module Foo {
  export var foo = 42;
  export function inc() { foo++; }
}
import foo from Foo;
Foo.inc();
foo;                 // 43
```

Figure 8. Changing the value of an exported name

```
var Foo = (function() {
  "use strict";
  var foo = 42;
  function inc() { foo++; }
  return Object.preventExtensions(Object.create(
    null,
    { foo: { get: function() { return foo; },
             enumerable: true },
      inc: { get: function() { return inc; },
             enumerable: true }
    }));
}).call(this);
var foo = Foo.foo;
Foo.inc();
foo;                 // 42
```

Figure 9. Translation of Figure 8 by Traceur

namic variable declaration does not have any effects; both of them may evaluate to `"bar"` if the dynamic variable declaration affects even the exported name. Or, the second reference to `bar` may evaluate to `"bar"` and `Foo.Bar.bar` may evaluate to `42`, if the dynamic variable declaration is in effect only in a module scope.

Finally, the proposal does not describe what happens when the evaluation of the `eval` function deletes variables that are exported to out of the module scope, as Figure 7 illustrates. Depending on the semantics, `Foo.Bar.foo` may evaluate to `"bar"` if the `eval` function call does not delete the variable `foo`, it may evaluate to `undefined` if the call deletes the variable, or it may throw an error if the semantics disallows such a delete operation.

Semantics in Traceur and TypeScript Based on the Harmony proposal, both Traceur and TypeScript translate an import declaration to an assignment expression, assigning the current value bound to the exported name to the imported name, which does not reflect any future changes of the value of the exported name. Figure 8 presents a simple module declaration of `Foo` exporting a variable `foo` and a function `inc`. The top-level imports `foo` from the module `Foo`, increments the value of `foo` by calling the exported function `inc`, and gets the value of the imported `foo`. According to the

```
var Foo;
(function (Foo) {
    Foo.foo = 42;
    function inc() { Foo.foo++; }
    Foo.inc = inc;
})(Foo || (Foo = {}));
var foo = Foo.foo;
Foo.inc();
foo;                 // 42
```

Figure 10. Translation of Figure 8 by TypeScript

semantics in the Harmony proposal, the value of `foo` should be `43`. However, both Traceur and TypeScript do not preserve the semantics. Figure 9 presents the translated version of the original code in Figure 8 by Traceur. Traceur uses the module pattern by creating an anonymous function, which uses the strict mode and returns a new object with two properties: `foo` and `inc`. It creates a new object by extending `Object` with `preventExtensions` to faithfully simulate read-only module instance objects. Similarly, Figure 10 presents the translated version of the original code by TypeScript. Unlike Traceur, TypeScript simply uses the traditional module pattern without any particular method for prohibiting future updates to the module object. Also, TypeScript uses a different concrete syntax for import declarations. More importantly, both of them produce "wrong" results `42`, because the translation is merely desugaring an import declaration to an assignment expression without maintaining a reference to the original exported name.

Interestingly, Traceur and TypeScript behave differently for the aforementioned examples. For the code example in Figure 6, Traceur evaluates both of them to `42`, but TypeScript evaluates them to `undefined` because their simple translation does not handle nested variable declarations correctly. Similarly for the code in Figure 7, Traceur throws a syntax error for trying to delete `foo` in strict mode, but TypeScript evaluates it to `"bar"`.

Semantics Compliant to Harmony Our formal semantics correctly preserves the semantics of the Harmony proposal. We provide a high-level explanation here and describe it formally in Section 3. We translate an export declaration to a pair of an exported name and its enclosing environment record, either a declarative environment record for a module scope or an object environment record for a module instance object, and we translate any references to the exported name to references to the name in the environment record. One might think that our translation is unnecessarily complicated and it is enough to translate an export declaration to simply a getter property in a module instance object rather than an assignment as in Traceur and TypeScript. Indeed, with such a translation, the value of the exported name would be always in synch with the value of the original variable. However, maintaining only a module instance object may lose connections to variables declared by `eval` function calls; it will refer to the new variable declared by `eval` calls rather than the variable declared in an enclosing scope. On the contrary, our translation mechanism always gets the correct value even in the presence of `eval` function calls.

An analogous problem arises with import declarations and we translate import declarations similarly to the export declarations. Instead of translating an import declaration to a property access in a module instance object, we translate an import declaration to a pair of an imported name and the module instance object that includes the imported entity, and we translate the references to the imported name to references to the property name in the module instance object, which always produces correct values.

```
foo();                              // "foo";
bar;                                // undefined;
function foo() { return "bar"; }
function foo() { return "foo"; }
var foo, bar = "foo", bar = "bar";
function bar() { return 42; }
bar;                                // "bar";
```

Figure 11. Name conflicts between functions and variables

```
function foo() { return "bar"; }
function foo() { return "foo"; }
function bar() { return 42; }
var foo, bar, bar;
foo();                              // "foo";
bar;                                // undefined;
bar = "foo";
bar = "bar";
bar;                                // "bar";
```

Figure 12. Same semantics for Figure 11

2.2 Name Conflict Resolution

Design Issue While one of the main goals of the module system is to avoid name conflicts, the Harmony proposal does not specify a name resolution mechanism precisely. A program may have various kinds of name conflicts between declarations in a program or a module: function declarations, variable declarations, module declarations, export declarations, and import declarations. The Harmony proposal describes a name conflict resolution mechanism between export declarations and between import declarations, but it does not describe any name conflict resolution mechanism between other kinds of declarations. To make things more complicated, the binding and evaluation order of declarations in a JavaScript program is different from their textual order.

Before describing name conflict resolution mechanisms, let us review the JavaScript semantics for evaluation of variable and function declarations again in more detail. As we discussed in Section 2.1, regardless of the textual order of function and variable declarations in a single scope, all the function declarations are evaluated first, all the variable declarations are bound and initialized to `undefined` next, and the remaining statements including the assignments from the variable declarations with initialization expressions, if any, are evaluated in their textual order. When there are name conflicts between function declarations, the function declaration that comes later in textual order overrides the other function declaration. Name conflicts between variable declarations have no effects in terms of name binding, but if both declarations have initialization expressions they are evaluated in order during the evaluation of the remaining statements. Similarly, when a function declaration and a variable declaration with an initialization expression declare the same name f, the function declaration is evaluated first and binds the name f and the initialization expression of the variable declaration overrides the function declaration.

For example, Figure 11 shows a JavaScript code with name conflicts between various declarations. According to the JavaScript semantics, the code example has the same semantics with the code example in Figure 12. The second function declaration overrides the first function declaration with the same name; variables are declared right after evaluating function declarations and initialized to `undefined`. Note that variable declarations do not override function declarations.

```
module Foo {
    export function toString() { return 42; }
}
module Bar {}
"" + {};                        // "[object Object]";
"" + Foo;                       // "42";
"" + Bar;                       // TypeError
```

Figure 13. Modules with and without `toString`

Semantics in Traceur and TypeScript Again, Traceur and TypeScript behave differently for name conflict resolution. Because Traceur uses the module pattern, it does not distinguish between module declarations and variable declarations with initialization expressions; a module declaration is translated to a variable declaration with an anonymous function expression as its initialization expression, and resolution of any name conflicts with it behaves just like for variable declarations following the JavaScript semantics. On the contrary, while TypeScript also uses the module pattern, it does not allow name conflicts with entities of different types; it syntactically rejects any name conflicts with different declarations.

Semantics Compliant to Harmony While the proposal does not provide a clear semantics for resolving name conflicts between various declarations including module declarations, we found that several options are incomparably reasonable in the sense that they are consistent with the JavaScript name resolution semantics. We may want to evaluate module declarations earlier than other declarations just like function declarations. We may also want to evaluate import and export declarations before other declarations; we may want to evaluate module, import and export declarations in any order, or we may want to evaluate module declarations before import and export declarations.

In our formalization of the module system, we chose to let one kind of declarations override other kinds of declarations: module declarations override the other kinds of declarations, import declarations override function and variable declarations, and function declarations override variable declarations. We made this design choice because we consider that module and import declarations are more important than local declarations, and a module declaration is more important than an import declaration which is simply an alias of a name in a module instance object. We do not argue that this design choice is the best option; rather, we provide our formalism as a starting point for any other researchers and developers to extend and modify according to their preferred design decisions.

2.3 Prototypes of Module Instance Objects

Design Issue While every JavaScript object has the `Object` prototype object as the top of its prototype chain, the Harmony proposal makes an exception for module instance objects:

> "A module instance object is a prototype-less object that provides read-only access to the exports of the module."

However, the unique characteristic of module instance objects may cause confusion when they are used with ordinary JavaScript objects, which does not satisfy one of the design goals of the Harmony module proposal: orthogonality from existing features.

Because `Object` provides default implementations for the `toString` and `valueOf` properties, every ordinary JavaScript object uses them unless some object in its prototype chain or the object itself provides its own implementations. For example, as Figure 13 shows, when we concatenate a string with an object, the object is implicitly converted to a string by calling the function bound to the `toString` property and concatenated to the given string. JavaScript developers use such implicit type conversions

heavily without worrying about the existence of the `toString` property, but module instance objects now break such programming. As in Figure 13, concatenation of a string to a module may result in a string or the `TypeError` exception depending on the existence of the `toString` property in the module instance object.

Semantics in Traceur and TypeScript Indeed, this feature is one of the sources to make differences between the module semantics of Traceur and TypeScript. While Traceur throws the `TypeError` exception for concatenating an empty string to the `Bar` module, TypeScript produces the `"[object Object]"` string.

Semantics Compliant to Harmony Our formalization of the module system is compliant to the Harmony proposal. However, we note that the proposed semantics in the proposal could be confusing to JavaScript developers as we described so far, and it might be a reasonable alternative to make module instance objects have the `Object` prototype object as their prototype object.

2.4 `eval` Function

Design Issue Though we used the `eval` function in our code examples so far, the Harmony proposal does not specify the interactions between the module system and the `eval` function clearly. While the proposal states that:

> "Reflective evaluation, via `eval` or the module loading API starts a new compilation and linking phase for the dynamically evaluated code."

it does not specify the semantics of module declarations, export declarations, and import declarations when they are evaluated by the `eval` function, which is not straightforward.

For example, whether the evaluation of an export declaration by the `eval` function affects the enclosing scope is debatable because it may require changes in the read-only module instance object that is already *sealed*. Since the `[[Extensible]]` internal property of a sealed object has the `false` value, we cannot add new properties to a sealed object. Also, since every property of a sealed object has the `false` value for its `[[Configurable]]` internal property, we cannot delete or modify the property. Therefore, prohibiting the evaluation of export and import declarations by the `eval` function from affecting the enclosing scope may be a reasonable option.

Semantics in Traceur and TypeScript Similarly for the case in Section 2.2, Traceur and TypeScript behave differently. Because module bodies are translated to strict code in Traceur, new declarations evaluated by the `eval` function do not affect the enclosing scope, and the `eval` function rejects any inputs of the extended syntax with the module system. On the contrary, TypeScript does not allow inputs of the extended syntax with the module system either but it does not signal any error.

Semantics Compliant to Harmony In our formalization, we chose to evaluate module, import, and export declarations by the `eval` function only when the function is called either in the global scope or in a module scope. Because module declarations can appear only at the top level of a program or a module body, we believe this design choice is consistent with the module proposal. If the `eval` function is called from a scope that is not the global scope or a module scope, evaluation of the function ignores any module, import, and export declarations in the input string.

3. Formalization of the JavaScript Module System

In this section, we describe the formalization of the JavaScript module system. Due to space limitations, we discuss only the central parts related to the design issues discussed in Section 2, and we refer the interested readers to our companion report [1].

Our formalization of the JavaScript module system translates JavaScript programs extended with the module system to plain JavaScript programs, which is based on our previous work [16] that translates JavaScript programs with modules to λ_{JS} programs. The contributions of the formalization in this paper over the previous work are as follows:

- It provides a semantics that is much closer to the JavaScript semantics. While λ_{JS} was proposed as "the essence of JavaScript," its semantics is very different from the JavaScript semantics. It is a simple extension of the lambda calculus, and the "desugaring" process is not really a syntactic simplification but rather a compilation. As the authors of λ_{JS} specified in their paper, type checking of translated λ_{JS} programs does not directly correspond to type checking of the original JavaScript programs. Although a small calculus may be useful for reasoning and proving some properties of JavaScript, the big gap between the source JavaScript language and the target λ_{JS} language makes it hard to understand and connect the relationships between original JavaScript programs and their translated λ_{JS} versions for JavaScript developers. Because the source-to-source translation in this paper describes the module semantics in terms of JavaScript, the language that the developers are already familiar with, it is more easily understandable.
- It enables more efficient and practical implementations. Because the generated λ_{JS} programs are huge and they manipulate large immutable objects, execution of λ_{JS} programs is impractically slow and it consumes a lot of memory. Instead, the source-to-source translation can take advantage of existing efficient JavaScript engines. For example, developers may write JavaScript programs with modules, translate them into plain JavaScript using the module rewriter described in Section 4, and run the translated programs on any JavaScript engine.
- It supports the `eval` function with one restriction. While λ_{JS} does not support the `eval` function at all, the source-to-source translation can utilize the ability of executing JavaScript code after translation. One restriction is that temporary helper function names should not conflict with the names introduced by the the `eval` function calls during module translation, which one can easily satisfy by using long strings of complicated characters for the helper function names.
- It is more flexible and adaptable in that it can address the design choices discussed in Section 2. Because the translation captures JavaScript-specific features in translated versions, it shows the design issues explicitly as we describe in Section 3.2.
- Because the formalization is built on top of the previous work [16], we can reuse much of the machinery developed there. For example, we reuse the module environment construction and the proofs of the validity checks.

3.1 JavaScript Module System

Our formalization describes the Harmony module proposal. Module declarations and import declarations can appear only at the top level of a program or a module body, and export declarations can appear only at the top level of a module body. The formalization builds on top of SAFE [17], which provides both a formal specification and implementation of JavaScript. We describe the module system using the Intermediate Representation (IR) syntax of SAFE; SAFE formally specifies compilation steps from JavaScript Abstract Syntax Tree (AST) to IR, and dynamic semantics of IR. We extend the syntax of JavaScript with module support as presented in Figure 14.

Before describing the translation from JavaScript with modules to plain JavaScript, let us review the semantics of the JavaScript

```
Program ::= SourceElement*
SourceElement ::= Statement
  | VariableDeclaration
  | FunctionDeclaration
  | ImportDeclaration
  | ModuleDeclaration
ModuleDeclaration ::= module Identifier { ModuleBody }
ModuleBody ::= ModuleElement*
ModuleElement ::= SourceElement
  | ExportDeclaration
ExportDeclaration ::= export ExportSpecifierSet
      (, ExportSpecifierSet)*;
  | export VariableDeclaration
  | export FunctionDeclaration
  | export get Identifier()
      { FunctionBody }
  | export set Identifier(Identifier)
      { FunctionBody }
ExportSpecifierSet ::= { ExportSpecifier
      (, ExportSpecifier)* }
      (from Path)?
  | Identifier (from Path)?
  | * (from Path)?
ExportSpecifier ::= Identifier (: Path)?
Path ::= Identifier(.Identifier)*
ImportDeclaration ::= import ImportClause
      (, ImportClause)*;
ImportClause ::= Path as Identifier
  | ImportSpecifierSet from Path
ImportSpecifierSet ::= { ImportSpecifier
      (, ImportSpecifier)* }
  | Identifier
ImportSpecifier ::= Identifier (: Identifier)?
```

Figure 14. Syntax of the module system

module system. When a JavaScript engine evaluates a program with modules, it first statically constructs a *module environment* which holds all the names in the global object and modules, and all the import and export relations. Then, it evaluates module, import, function, and variable declarations, and statements in order. When it evaluates module declarations, it first instantiates every module by constructing a *module scope* and a *module instance object*. Each module declaration introduces a new scope called a module scope, and its module body is evaluated in the module scope. An evaluated module called a module instance object contains lexically encapsulated members and exported bindings. An evaluation of a module results in a JavaScript object that reflects the exported bindings of a module instance object. After instantiating the modules, a JavaScript engine initializes all the module bindings and makes all the module instance objects read-only. Initializing a module binding is to evaluate the statements in the module body in its module scope. For mutually recursive import statements, function, variable, and nested module declarations are evaluated in the module scope, and the getters for the exported names are set in the module instance object. Finally, it seals all the module instance objects to make them read-only and substitutes each imported name with its canonical name of which the imported name is an alias.

$$(\Sigma, \phi), p \rightarrow_m \mathit{MDecl}[\![p]\!](\Sigma, \phi)$$

$$\mathit{MDecl}[\![s_1 \cdots s_n]\!](\Sigma, \phi) = \mathit{MDecl}[\![s_1]\!](\Sigma, \phi) \cdots \mathit{MDecl}[\![s_n]\!](\Sigma, \phi)$$

$$\begin{aligned}
&\mathit{MDecl}[\![\texttt{module}\ M\ \{s_1 \cdots s_n\}]\!](\Sigma, \phi) = \\
&\quad \texttt{var}\ M = \texttt{new}\ (\texttt{function(f, p)}\ \{ \\
&\qquad\qquad \texttt{f.prototype} = \texttt{p}; \\
&\qquad\qquad \texttt{return f}; \\
&\qquad\quad \}\ (f, p))\ (); \\
&\quad \texttt{Object.seal}(M);
\end{aligned}$$

where $f =$ `function() {`

$\quad \mathit{QName}[\![\mathit{FDecl}[\![s_1 \cdots s_n]\!](\Sigma, \phi')]\!](\mathit{QNEnv}(\Sigma, \phi'))$

$\quad \mathit{QName}[\![\mathit{VDecl}[\![s_1 \cdots s_n]\!](\Sigma, \phi')]\!](\mathit{QNEnv}(\Sigma, \phi'))$

$\quad \mathit{QName}[\![\mathit{MDecl}[\![s_1 \cdots s_n]\!](\Sigma, \phi')]\!](\mathit{QNEnv}(\Sigma, \phi'))$

$\quad \mathit{QName}[\![\mathit{Exports}[\![s_1 \cdots s_n]\!](\Sigma, \phi')]\!](\mathit{QNEnv}(\Sigma, \phi'))$

$\quad \texttt{this.update}M = \texttt{function(arguments)}\ \{$

$\qquad \mathit{QName}[\![\mathit{Others}[\![s_1 \cdots s_n]\!](\Sigma, \phi')]\!]$

$\qquad\quad (\mathit{QNEnv}(\Sigma, \phi'))$

$\quad \};$

`}`

$$\text{and } p = \begin{cases} \texttt{Object.prototype} & \text{if } \phi = \epsilon \\ \phi & \text{otherwise} \end{cases}$$

$$\text{and } \phi' = \begin{cases} M & \text{if } \phi = \epsilon \\ \phi.M & \text{otherwise} \end{cases}$$

$$\mathit{MDecl}[\![d]\!](\Sigma, \phi) = ; \qquad \text{if } d \notin \mathit{ModuleDeclaration}$$

Figure 15. Module translation

Translation of a program p consists of several phases as follows:

$$\frac{(\Sigma, \phi), p \rightarrow_f p_f \qquad (\Sigma, \phi), p \rightarrow_v p_v \qquad (\Sigma, \phi), p \rightarrow_m p_m \qquad (\Sigma, \phi), p \rightarrow_u p_u \qquad (\Sigma, \phi), p \rightarrow_s p_s}{(\Sigma, \phi), p \rightarrow p_f\ p_v\ p_m\ p_u\ p_s}$$

Before translating the program, we build a module environment Σ, which maps all the names in p to their unique qualified names, as described in the previous work [16]. For given Σ and a path from the top-level ϕ, which is the empty path ϵ for p, we first translate declarations: function declarations by the $\rightarrow_f$ rule, variable declarations by the $\rightarrow_v$ rule, and module declarations by the $\rightarrow_m$ rule. Then, we initialize module instance objects by the $\rightarrow_u$ rule, and rename all the imported names in top-level variable declarations, function declarations, and statements with their corresponding exported names by the $\rightarrow_s$ rule.

Translation of a module declaration uses the module pattern. As Figure 15 describes, for given Σ and ϕ, translation of a module declaration "`module` M $\{s_1 \cdots s_n\}$" uses a function scope to represent a module scope. Because any function value is an object in JavaScript, we translate a module declaration to a variable declaration initialized with the `new` statement using a function as a constructor. Specifically, in Figure 15, the declaration of a module M is translated to the declaration of a variable M which constructs a function object.[2] The function sets up its prototype object, which we discuss in Section 3.2. Then, it returns a module instance object f with exported names as its properties; in the body

[2] While the function translates its `arguments` variable as a pair of a getter and a setter to avoid name conflicts, we omit the translation for presentation brevity.

$$(\Sigma, \phi), p \rightarrow_f \mathit{FDecl}[\![p]\!](\Sigma, \phi)$$

$$\mathit{FDecl}[\![s_1 \cdots s_n]\!](\Sigma, \phi) = \mathit{FDecl}[\![s_1]\!](\Sigma, \phi) \cdots \mathit{FDecl}[\![s_n]\!](\Sigma, \phi)$$

$$\mathit{FDecl}[\![(\texttt{export})\ \texttt{function}\ f(x_1, \cdots, x_n)\ \{s_1 \cdots s_k\}]\!](\Sigma, \phi) = \texttt{function}\ f(x_1, \cdots, x_n)\ \{s_1 \cdots s_k\}$$

$$\mathit{FDecl}[\![d]\!](\Sigma, \phi) = ; \quad \text{if } d \notin \mathit{FunctionDeclaration}$$

Figure 16. Function translation

$$(\Sigma, \phi), p \rightarrow_v \mathit{VDecl}[\![p]\!](\Sigma, \phi)$$

$$\mathit{VDecl}[\![s_1 \cdots s_n]\!](\Sigma, \phi) = \texttt{var}\ x_1, \cdots, x_k \quad \text{where } x_i \in \mathit{VName}[\![s_1 \cdots s_n]\!](\Sigma, \phi)\ \wedge\ 1 \le i \le k$$

Figure 17. Variable translation

$$\mathit{Others}[\![s_1 \cdots s_n]\!](\Sigma, \phi) = \mathit{Others}[\![s_1]\!](\Sigma, \phi) \cdots \mathit{Others}[\![s_n]\!](\Sigma, \phi)$$

$$\mathit{Others}[\![(\texttt{export})\ \texttt{var}\ x_1(= e_1), \cdots, x_n(= e_n)]\!](\Sigma, \phi) = x_{i_1} = e_{i_1}; \cdots x_{i_k} = e_{i_k}; \quad \text{where } \exists\, e_{i_j} \wedge\ i_1 \le i_j \le i_k$$

$$\mathit{Others}[\![s]\!](\Sigma, \phi) = s \quad \text{if } s \in \mathit{Statement}$$

$$\mathit{Others}[\![d]\!](\Sigma, \phi) = ; \quad \text{if } d \notin (\mathit{VariableDeclaration} \cup \mathit{Statement})$$

Figure 18. Other declaration translation

of the function object f, the declarations and statements are reordered as we discuss in Section 3.2. Function declarations come first, if any, variable declarations, sub-module declarations, export declarations, and statements, if any, follow in order. As Figure 16 describes, *FDecl* collects function declarations including exported ones without the `export` keyword, and *VDecl* shown in Figure 17 collects variable names in M by using the helper function *VName* and declares them. Note that any initialization expressions in variable declarations are deferred to statements. All the names in each function, variable, and module declarations in M are replaced with their unique qualified names by using the helper functions *QName* and *QNEnv*. We omit the definitions of the helper functions in this paper due to space limitations, and refer the interested readers to our companion report [1]. As we discuss in Section 3.2, *Exports* makes getters for exported names. Because of nested modules and mutually-dependent modules, the translation resolves names that appear in declarations and statements in different scopes at different phases. It creates a temporary function `update`M to replace the names in the remaining declarations and statements with their unique qualified names after constructing module instance objects; the $\rightarrow_u$ rule calls this function to initialize module instance objects. We assume that the translator chooses `update`M as a non-conflicting name with any names in the program. Figure 18 presents *Others* which collects the initialization expressions in variable declarations and the statements in M. We make the module instance object read-only by calling `Object.seal`, which finishes translation of a module declaration.

Now that all module instance objects finished declaration of their functions, variables, and sub-modules, the $\rightarrow_u$ rule presented in Figure 19 initializes each module instance object recursively; it calls the temporary function `update`M to perform renaming in the module scope, deletes the temporary function object `update`M,

$$\frac{(\Sigma, \phi), s_i \rightarrow_u s_i' \qquad 1 \le i \le n}{(\Sigma, \phi), s_1\ \cdots\ s_n \rightarrow_u s_1'\ \cdots\ s_n'}$$

$$\frac{(\Sigma, \phi.M), s_i \rightarrow_u s_i' \qquad 1 \le i \le n}{(\Sigma, \phi), \texttt{module}\ M\ \{s_1\ \cdots\ s_n\} \rightarrow_u \begin{array}{l}\phi.M.\texttt{update}M();\\ \texttt{delete}\ \phi.M.\texttt{update}M;\\ \texttt{Object.freeze}(\phi.M);\\ s_1'\ \cdots s_n'\end{array}}$$

$$\frac{d \notin \mathit{ModuleDeclaration}}{(\Sigma, \phi), d \rightarrow_u ;}$$

Figure 19. Module initialization

$$\frac{\{i_1, \cdots, i_k\} = \{i \mid s_i \notin \mathit{ModuleDeclaration} \wedge\ s_i \notin \mathit{ImportDeclaration}\ \wedge 1 \le i \le n\}}{(\Sigma, \phi), s_1 \cdots s_n \rightarrow_s \mathit{QName}[\![s_{i_1} \cdots s_{i_k}]\!](\mathit{QNEnv}(\Sigma, \phi))}$$

Figure 20. Module renaming

$$\mathit{Exports}[\![s_1 \cdots s_n]\!](\Sigma, \phi) = \mathit{Exports}[\![s_1]\!](\Sigma, \phi) \cdots \mathit{Exports}[\![s_n]\!](\Sigma, \phi)$$

$$\mathit{Exports}[\![\texttt{export}\ \{x\}]\!](\Sigma, \phi) = \mathit{Exports}[\![\texttt{export}\ \{x\colon x\}]\!](\Sigma, \phi)$$

$$\mathit{Exports}[\![\texttt{export}\ \{x\colon \phi'\}]\!](\Sigma, \phi) = \mathit{Export}(\texttt{get}, x, ()\{\texttt{return}\ \mathit{lookup}[\![\phi']\!](\Sigma, \phi);\})$$

$$\mathit{Exports}[\![\texttt{export get}\ f()\{s_1 \cdots s_n\}]\!](\Sigma, \phi) = \mathit{Export}(\texttt{get}, f, ()\{\mathit{Exports}[\![s_1]\!](\Sigma, \phi) \cdots \mathit{Exports}[\![s_n]\!](\Sigma, \phi)\})$$

$$\mathit{Exports}[\![\texttt{export set}\ f(x)\{s_1 \cdots s_n\}]\!](\Sigma, \phi) = \mathit{Export}(\texttt{set}, f, (x)\{\mathit{Exports}[\![s_1]\!](\Sigma, \phi) \cdots \mathit{Exports}[\![s_n]\!](\Sigma, \phi)\})$$

$$\mathit{Export}(\mathit{accessor}, f, \mathit{ftn}) =$$
```
var $desc =
    Object.getOwnPropertyDescriptor(this, "f");
delete this.f;
if (typeof $desc == "undefined")
  $desc = {configurable : true};
$desc.accessor = function ftn;
Object.defineOwnProperty(this, "f", $desc);
```

Figure 21. Export translation (partial)

and freezes the module instance object. Finally, the $\rightarrow_s$ rule presented in Figure 20 renames all the imported names in top-level variable declarations, function declarations, and statements with their corresponding exported names.

3.2 Formal Specification of the Design Issues

Let us revisit the design issues in the JavaScript module system discussed in Section 2, and explain the issues and our design decisions in the context of the formal specification.

Import and Export Declarations To correctly preserve the semantics of the import and export declarations in the Harmony proposal, we translate an export declaration to a pair of an exported name and its enclosing environment record, and we translate any

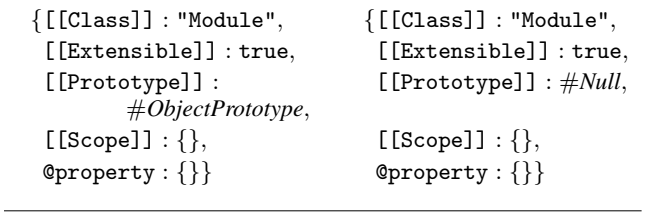

Figure 22. Module instance objects with/without prototype object

references to the exported name to references to the name in the environment record. In the formalization, we represent a pair of an exported name and its enclosing environment record as an accessor property in a module instance object. As Figure 21 describes, an exported name x_i in a module instance object ϕ is translated to a getter named x_i of the object, where $lookup[\![\phi']\!](\Sigma, \phi)$ returns the unique qualified name denoted by ϕ'. We show translation of some export declarations in Figure 21 for presentation brevity; translation of other export declarations is very similar. Then, any reference to x_i becomes a getter call on the module instance object ϕ, which preserves the semantics even when the `eval` function call creates a new value with the same name x_i. Similarly, references to imported names become getter calls on module instance objects, which preserves the semantics even in the presence of `eval` function calls.

Name Conflict Resolution As we discussed in Section 2.2, the proposal does not provide a clear semantics for resolving name conflicts between various declarations. While several options are incomparably reasonable, we chose a specific precedence so that module declarations override the other kinds of declarations and function declarations override variable declarations. Figure 15 describes this precedence in the body of f. We made this design choice because we consider that module declarations are more important than function or variable declarations. One can make different design decisions by changing the order of declarations in the body of f.

According to the Harmony proposal, the translation should instantiate and initialize modules in their textual order. At the same time, because the proposal allows nested modules and mutually-dependent modules, we may not have all the information to resolve names by a single scan of a program. Thus, name resolution for the JavaScript module system requires several steps: we first build a module environment Σ to collect all the names, create module instance objects possibly uninitialized, and finish initialization and freeze all the module instance objects.

Prototypes of Module Instance Objects As we discussed in Section 2.3, the proposed semantics of the prototypes of module instance objects in the proposal could be confusing or rather unintuitive to JavaScript developers. As a reasonable alternative semantics, in the formalization in Figure 15, the function object representing a module scope takes p as its second argument, which denotes its prototype object. If a module is declared at the top-level of a program, its module instance object sets its prototype object with `Object.prototype`. Otherwise, a module instance object sets its prototype object with its enclosing module instance object. For a compliant semantics to the Harmony proposal, one may not set the prototype object of a module instance object.

More concretely, Figure 22 shows module instance objects with the empty scope and the empty property as placeholders. The value of the `[[Class]]` internal property is `"Module"` and the value of the `[[Extensible]]` internal property is `true` until the objects are sealed. The left module instance object represents the semantics where its prototype object is `Object.prototype` just like other JavaScript objects and the right module instance object represents

$$\frac{\Gamma = (H, A, tb, \Sigma, \phi) \qquad (v_1, v_3) = isEval(\Gamma, y, z_1, z_2) \qquad inModScope(H, A) \qquad \Sigma' = Env[\![v_3]\!](\Sigma, \phi) \qquad (H, A, tb, \Sigma', \phi), x = \texttt{eval}(v_1, v_3) \rightarrow_e (H', A'), ct}{\Gamma, x = y(z_1, z_2) \rightarrow_e (H', A'), ct}$$

$$\frac{\Gamma = (H, A, tb, \Sigma, \phi) \qquad (v_1, v_3) = isEval(\Gamma, y, z_1, z_2) \qquad \neg\, inModScope(H, A) \qquad v_4 = RemoveModule(v_3) \qquad \Gamma, x = \texttt{eval}(v_1, v_4) \rightarrow_e (H', A'), ct}{\Gamma, x = y(z_1, z_2) \rightarrow_e (H', A'), ct}$$

$$\frac{\Gamma = (H, A, tb, \Sigma, \phi) \qquad \Gamma, y \rightarrow_e \#GlobalEval \qquad \Gamma, z_1 \rightarrow_e v_1 \qquad v_1 \in Loc \qquad \Gamma, z_2 \rightarrow_e v_2}{isEval(\Gamma, y, z_1, z_2) = (v_1, ToString(H, v_2))}$$

$$\frac{}{inModScope(H, \#Global)}$$

$$\frac{l \in dom(H) \qquad H(l) \in Object \qquad H(l).[[\texttt{Class}]] = \text{“}\texttt{Module}\text{”} \qquad H(l).[[\texttt{Scope}]] = A}{inModScope(H, A)}$$

Figure 23. Evaluation of the `eval` function

the semantics where it does not have its prototype object as the Harmony proposal specifies. In Figure 22, *#ObjectPrototype* denotes the location of the `Object` prototype object and *#Null* denotes that the object does not have its prototype object.

`eval` Function Because module declarations can appear only at the top level of a program or a module body, our formalization chose to evaluate module, import, and export declarations by the `eval` function only when the function is called either in the global scope or in a module scope. Figure 23 describes the semantics of the `eval` function call.

The first rule describes the case when the `eval` function is called in the global scope or in a module scope. Under the evaluation context Γ, which consists of a heap H, an execution environment A, the value of `this` *tb*, a module environment Σ, and a path from the top level ϕ, it first checks whether the function being called is the global `eval` function by *isEval*. The third rule defines *isEval*, which makes sure that the global `eval` function denoted by *#GlobalEval* is being called, and returns a pair of valid arguments to the function. The fourth and the fifth rules define *inModScope*, which checks whether the function is being called in the global scope or in a module scope. When the `eval` call is *inModScope*, it updates the module environment by *Env* to treat the string argument of the `eval` function just like a JavaScript code and evaluates it. After evaluating the function call, it produces (H', A'), a heap and an execution environment that may be updated during the evaluation, and the function call result *ct*.

The second rule describes the case when the `eval` function is called locally. Then, it removes any module-related declarations from the input string by *RemoveModule* and evaluates it. We omit the definitions of the helper functions for presentation brevity.

As an alternative semantics, one may want to allow evaluation of the `eval` function to declare modules, imports, and exports. We can support such a semantics in the formalization by replacing the

first and the second rules with the following:

$$\frac{\Gamma = (H, A, tb, \Sigma, \phi) \qquad (v_1, v_3) = isEval(\Gamma, y, z_1, z_2) \qquad \Sigma' = Env[\![v_3]\!](\Sigma, \phi) \qquad (H, A, tb, \Sigma', \phi), x = \texttt{eval}(v_1, v_3) \rightarrow_e (H', A'), ct}{\Gamma, x = y(z_1, z_2) \rightarrow_e (H', A'), ct}$$

In other words, the alternative semantics evaluates the string argument of the `eval` function without checking whether it is being called in the global scope or in a module scope by *inModScope*.

3.3 Validity Properties of the JavaScript Module System

Because our formalization is built on top of the previous work [16], we can reuse the machinery used to prove the properties of valid programs. For a valid JavaScript program p that may declare and use modules, its translated program in JavaScript does not include any free variables.

Property 1 (Validity of Translation Rules)**.** *For a valid program p, evaluating its translated version does not cause any error due to free variables.*

Proof Sketch. Let $FV(s)$ denote the set of the free variables in a statement s. For each evaluation rule:

$$\forall H, A, tb, \Sigma, \phi, s.$$
$$(H, A, tb, \Sigma, \phi), s \rightarrow_e (H', A'), ct \text{ for some } H', A', ct$$

we show that the evaluation does not add new free variables. Assume that the property is false, that is, the translated version p' causes an error due to a free variable. By the properties of free variables and because evaluations do not add new free variables, $FV(p')$ includes the free variable causing the error during evaluation of p'. Because p is a valid program, we can show that $FV(p) = \emptyset$, which makes the assumption false. Thus, the translated version of a valid program does not cause an error due to a free variable. □

In a valid program p, the module scopes are correctly *isolated* from outside.

Property 2 (Isolation of Module Scopes)**.** *When a module M declares a variable x but does not export it, the module-scope variable x does not affect outside its enclosing module M.*

Proof Sketch. Let $p(v)$ be `module` M `{var` x `=` v`;}` p. By establishing the exact module environments for p and $p(v)$, we can show that the module environment of $p(v)$ is a simple extension of the module environment of p. We can also show that two environments $Env[\![p]\!]$ and $Env[\![p(v)]\!] \backslash Env[\![p]\!]$ are disjoint. Therefore, we can show that any translation of $p(v)$ shares a similar structure that is independent of the choice of v. By showing that the translation of p by the $\rightarrow_s$ rule under the module environment Σ derived from the translation of $p(v)$ does not contain any occurrence of $M.x$, we can finish the proof. □

4. Implementation of the JavaScript Module System

We implemented the JavaScript module system on top of SAFE [17] as a source-to-source translator and made it publicly available [15]. Figure 24 illustrates a high-level architecture of the important parts of SAFE. The architecture accepts a JavaScript program, translates it through AST, IR, and Control Flow Graph (CFG), each of which is utilized by various analyses and tools. For example, AST suites best for clone detection and pretty printing the original JavaScript program; IR works best for evaluation and test data generation; and CFG works best for type-based analysis and bug detection among others. For presentation brevity, we omit other features of SAFE such as Web IDL [26] supports. Figure 25 presents the extended architecture of SAFE with support for the module system. The dotted boxes denote extended parts to support the module system, and the *ModuleRewriter* phase translates an AST that may include module syntax into another AST that does not include any module syntax. Due to space limitations, we omit the details of *ModuleRewriter*, but discuss its properties and limitations. JavaScript engine developers can evaluate their implementations and ours to articulate and discuss ambiguous semantics from the Harmony proposal. Because our implementation is open sourced, the developers can apply our technique to their engines. Or, they can even use our *ModuleRewriter* as a preprocessor to support the module system as a sort of macros.

As we discussed in Section 3.1, the translation mechanism is based on the module pattern. Because translation of a module consists of instantiation and initialization, the control should reenter the function scope at initialization, which is not available with JavaScript objects. Therefore, our implementation creates a temporary object in the global scope which has temporary functions created from the module scope at instantiation time, and it calls the temporary functions to initialize the modules at initialization time. We implement import statements by replacing the references to imported names with the corresponding property accesses to the module instance objects. Figure 26 presents a translated code of the example in Figure 8 in plain JavaScript by *ModuleRewriter*.

The limitations of *ModuleRewriter* are due to the quirky dynamic semantics of JavaScript: the `eval` function and the `with` statement. First, *ModuleRewriter* does not handle the references to imported names in a string passed to the `eval` function as an argument. Because *ModuleRewriter* statically collects the references to imported names and replaces them to corresponding property accesses to module instance objects at compile time, it should be able to collect the references to imported names from a string at compile time. Secondly, the `with` statement takes an object, called a *with object*, and introduces the properties of the with object to the environment at run time, which basically supports dynamic scoping. The properties of the with object shadow any bindings with the same name in the enclosing context. Therefore, a reference to an identifier may evaluate to an entity defined in the enclosing context or to a property in the with object. Because it is not possible to determine whether an identifier refers to an imported name at compile time, it is not possible to replace identifiers in the `with` statement. Thanks to Park *et al.* [20], we can desugar most of the `with` statements at compile time; for example, we can desugar the following `with` statement:

```
with (o) { x = 42; }
```

to the following code without using the `with` statement:

```
("x" in o ? o.x : x) = 42;
```

preserving the semantics. Indeed, the SAFE architecture includes the `with` rewriter implementation by default. Therefore, *ModuleRewriter* can handle the `with` statements unless they call dynamic code generation functions such as the `eval` function.

5. Related Work

The ECMAScript Harmony proposal [2] describes the new features adopted in the next release of ECMAScript such as modules, classes, generators, and block scoped bindings. Most of the proposal have already been incorporated into the ECMAScript 6 language specification draft, but the module proposal [10] is not yet included. The module proposal provides a brief, high-level description of the JavaScript module system but the informal specification in prose does not explain the module semantics precisely.

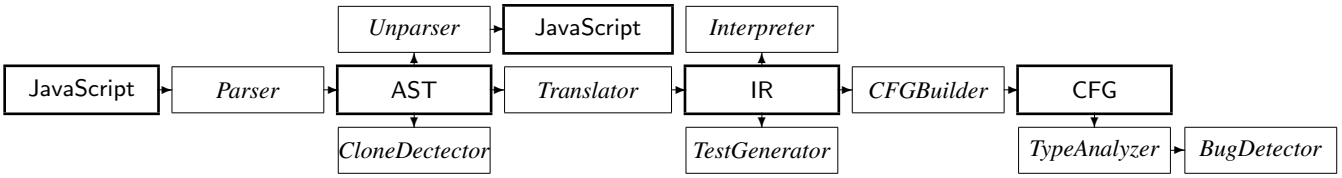

Figure 24. SAFE architecture

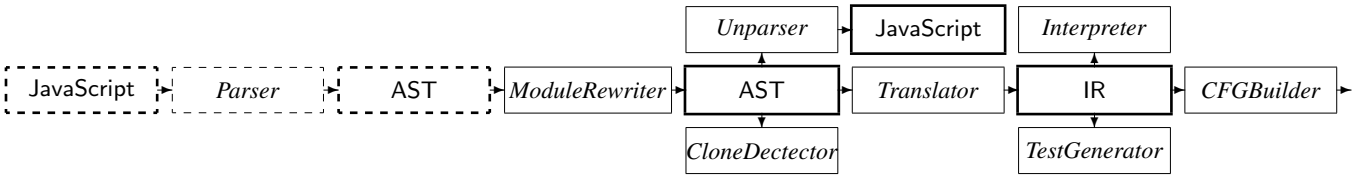

Figure 25. SAFE architecture with module support

Kang and Ryu [16] designed a formal specification and implementation of a JavaScript module system based on the Harmony proposal. They formalized and implemented their module system using λ_{JS} [9], which is a core calculus of ECMAScript 3. While they provide a small set of conventional formal semantics rules of λ_{JS} and translation rules from JavaScript to λ_{JS}, the implementation is impractically slow. They described the formal semantics of the module system as a set of translation rules from JavaScript with modules to λ_{JS}, and they implemented the module system by extending the translation rules. While the formalization and implementation are solid and rigorous, the practicality of their approach is not clear. Because the semantics of λ_{JS} is very much different from the original JavaScript, the big gap between JavaScript and λ_{JS} makes it hard for JavaScript developers to understand and connect the relationships between original JavaScript programs and their translated λ_{JS} versions. Because the generated λ_{JS} programs are huge, executing λ_{JS} programs has been absurdly slow. On the contrary, our formalization is similarly rigorous but much easier to understand because it is a source-to-source translation, and our implementation is much more efficient and practical.

Traceur [6] is a compiler developed by Google that compiles JavaScript.next code into one in ECMAScript 5. While their implementation is written in JavaScript of size more than 20,000 lines, they do not provide any formal specification or a detailed description of the system. As we have seen in this paper, the module implementation in Traceur does not satisfy the semantics in the Harmony module proposal in many cases.

TypeScript is a superset of JavaScript that supports a type system for static checking and verification. It also provides some features of JavaScript.next including the module system, and it compiles a TypeScript program into a plain JavaScript program. Unlike Traceur, supporting JavaScript.next closely today is not one of the main goals of TypeScript. It focuses more on type inference and type annotation for better static checking and analysis. Thus, its module support is more premature than Traceur, often leads to different semantics from the Harmony module proposal. It supports somewhat different module syntax and it does not support mutually recursive imports, for example.

While both Traceur and TypeScript support very premature module systems with different semantics from the Harmony proposal without any detailed description of them, our module system is formally specified and implemented preserving the semantics of the Harmony proposal. Our implementation is built on top of SAFE, a scalable analysis framework for JavaScript, which provides various facilities for JavaScript program manipulation. Because the implementation of the SAFE architecture uses tools to automatically generate AST nodes by ASTGen [22] and parsers by Rats! [7], it is easily adaptable to any syntax changes and extensions. The various components of SAFE such as the unparser to pretty print plain JavaScript syntax, the interpreter to evaluate JavaScript programs, and the `with` rewriter to desugar the `with` statements away have greatly improved the quality and productivity of our implementation.

6. Conclusion

JavaScript is now one of the most widely used programming languages from small scripts embedded in web documents to large stand-alone projects such as web applications and compilers. The lack of any language-level module system in JavaScript has caused many problems and an ad-hoc programming pattern, which finally introduces a language-level module system to JavaScript in the next release of the ECMAScript language specification. Even though the ECMAScript Harmony proposal for the next release includes a JavaScript module system, its current status is not yet ready to be incorporated into the language specification draft. Its brief, informal description of the module system does not specify the module semantics precisely, which often leads to ambiguous or unclear interactions with existing language features. Thus, even Traceur from Google and TypeScript from Microsoft do not support the module system semantics correctly yet.

In this paper, we investigate and describe design issues with the JavaScript module system with concrete code examples. We explain each issue using the results from Traceur and TypeScript and discuss reasonable design decisions for the issues. We present a formal specification of the module system as a source-to-source transformation from JavaScript with modules to plain JavaScript. We also provide an open-source implementation of the module system so that JavaScript developers can use the future module system in advance and JavaScript engine developers can take advantage of our implementation techniques. We believe that our discussion on the design issues of the module system and its formal specification and implementation will help the designers of the Harmony proposal and the developers of the proposal to understand the module system more clearly. Because our module system is not particularly tied to JavaScript, we expect that it will be applicable to other scripting languages such as Python and Ruby.

```
var __initarg = {},
    __initfun = {},
    __extmod = {},
    __intmod = {},
    __Object = Object,
    Foo, foo;
Foo = __extmod.Foo = new (function(arguments) {
  function inc() { foo ++; }
  var inc, foo;
  __intmod.Foo = {
      get inc() { return inc; },
      get foo() { return foo; }
  };
  (function (__this) {
    var __desc =
      __Object.getOwnPropertyDescriptor(__this, "foo");
    delete __this.foo;
    if(typeof __desc == "undefined")
      __desc = { configurable : true };
    __desc.get = (function foo() {
      return __intmod.Foo.foo;
    });
    __Object.defineProperty(__this, "foo", __desc);
  })(this);
  (function (__this) {
    var __desc =
      __Object.getOwnPropertyDescriptor(__this, "inc");
    delete __this.inc;
    if(typeof __desc == "undefined")
      __desc = { configurable : true };
    __desc.get = (function inc() {
      return __intmod.Foo.inc;
    });
    __Object.defineProperty(__this, "inc", __desc);
  })(this);
  __initfun.Foo = (function(arguments) {
      foo = 42;
  });
  __initarg.Foo = {
      get arguments() { return arguments; },
      set arguments(x) { arguments = x; }
  };
})({
  get arguments() { return arguments; },
  set arguments(x) { arguments = x; }
   });
__Object.seal(Foo);
__initfun.Foo.call(this, __initarg.Foo);
Foo.inc();
__intmod.Foo.foo;
```

Figure 26. Translation of Figure 8 by *ModuleRewriter*

Acknowledgments

This work is supported in part by Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grants NRF-2011-0016139 and NRF-2008-0062609), Samsung Electronics, S-Core, and Google.

References

[1] Junhee Cho and Sukyoung Ryu. JavaScript module system: Exploring the design space (extended with proofs). `http://plrg.kaist.ac.kr/research/publications`.

[2] ECMA. Harmony proposals. `http://wiki.ecmascript.org/doku.php?id=harmony:proposals`.

[3] ECMA. Harmony proposals – draft specification for ES.next (Ecma-262 Edition 6). `http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts`.

[4] ECMA. *ECMA-262: ECMAScript language specification*. 5th edition, December 2009.

[5] Linux Foundation. Tizen: an open source, standards-based software platform for multiple device categories. `https://www.tizen.org`.

[6] Google. Traceur-compiler: Google's vehicle for JavaScript language design experimentation. `https://code.google.com/p/traceur-compiler/`.

[7] Robert Grimm. Rats! – an easily extensible parser generator. `http://cs.nyu.edu/~rgrimm/xtc/rats.html`.

[8] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 20th Internaional Symposium on Software Testing and Analysis*, 2011.

[9] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, 2010.

[10] Dave Herman and Sam Tobin-Hochstadt. Harmony proposals – modules. `http://wiki.ecmascript.org/doku.php?id=harmony:modules`.

[11] Alexa Internet. Alexa. `http://www.alexa.com`.

[12] Joynet. Node.js. `http://nodejs.org/`.

[13] jQuery Foundation. jQuery. `http://jquery.com/`.

[14] jQuery Foundation. `jQuery.noConflict()`. `http://api.jquery.com/jQuery.noConflict/`.

[15] PLRG @ KAIST. SAFE: Scalable Analysis Framework for ECMAScript. `http://safe.kaist.ac.kr`.

[16] Seonghoon Kang and Sukyoung Ryu. Formal specification of a JavaScript module system. In *Proceedings of the 2012 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2012.

[17] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *Proceedings of the 19th International Workshop on Foundations of Object-Oriented Languages*, 2012.

[18] Microsoft. TypeScript. `http://www.typescriptlang.org`.

[19] Eric Miraglia. A JavaScript module pattern. `http://www.yuiblog.com/blog/2007/06/12/module-pattern/`.

[20] Changhee Park, Hongki Lee, and Sukyoung Ryu. All about the `with` statement in JavaScript: Removing `with` statements in JavaScript applications. In *Proceedings of the Dynamic Language Symposium 2013*, 2013.

[21] Samsung Electronics. Samsung Smart TV. `http://developer.samsung.com/smarttv`.

[22] Brian R. Stoler, Eric Allen, and Dan Smith. ASTGen. `http://sourceforge.net/projects/astgen`.

[23] Prototype Core Team. Prototype. `http://prototypejs.org/`.

[24] The MooTools Dev Team. MooTools: a compact JavaScript framework. `http://mootools.net`.

[25] W3C. Document Object Model (DOM). `http://www.w3.org/DOM/`.

[26] W3C. Web IDL. `http://www.w3.org/TR/WebIDL`.