JSPP: Morphing C++ into JavaScript

Christopher Chedeau, Didier Verna
EPITA Research and Development Laboratory
14–16, rue Voltaire
94276 Le Kremlin-Bicêtre CEDEX, France
{christopher.chedeau,didier.verna}@lrde.epita.fr

ABSTRACT

In a time where the differences between static and dynamic languages are starting to fade away, this paper brings one more element to the "convergence" picture by showing that thanks to the novelties from the recent C++0x standard, it is relatively easy to implement a JavaScript layer on top of C++. By that, we not only mean to implement the language features, but also to preserve as much of its original notation as possible. In doing so, we provide the programmer with a means to freely incorporate highly dynamic JavaScript-like code into a regular C++ program.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classification—*multi-paradigm languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages

Keywords

C++, JavaScript, Multi-Paradigm Programming

1. INTRODUCTION

The relations between static and dynamic language communities are notoriously tense. People from both camps often have difficulties communicating with each other and tend to become emotional when it comes to such concerns as expressiveness, safety or performance. However, and perhaps due (at least partially) to the constantly increasing popularity of modern dynamic languages such as Ruby, Python, PHP or JavaScript, those seemingly irreconcilable worlds are starting to converge. For example, while C# recently provided support for dynamic types, Racket[8], a descendant from Scheme, allows you to freely mix static typing with an otherwise dynamically typed program.

C++0x[13], the new standard for the C++ language, incorporates many concepts well known in the functional or dynamic worlds, such as "lambda (anonymous) functions", "range-based iterators" (an extended iteration facility) and "initializer lists" (an extended object initialization facility).

To bring one more element to this general trend towards convergence, this article relates the surprising discovery that given this new C++ standard, it is relatively easy to implement JavaScript in C++. The idea is to make all the features of JavaScript available in C++ while preserving the original JavaScript syntax as much as possible. Ultimately, we would like to be able to simply copy and paste JavaScript code in a C++ source file and compile it as-is.

A first prototypical implementation of a syntax-preserving JavaScript engine in C++ has been developed. The project is called JSPP. JSPP currently implements all the core features of JavaScript in about 600 lines of code. Being only a prototype however, JSPP is not currently optimized and does not provide the JavaScript standard libraries. JSPP is an open-source project. The source code and some examples can be found at the project's Github Repository[1].

In section 2, we describe the implementation of JavaScript's dynamic type system and object notation (*a.k.a.* JSON[4]). Section 3 on page 3 provides an explanation on how to use the C++0x lambda functions to emulate JavaScript functions and object constructors. Section 4 on page 5 addresses the support for JavaScript object properties (most notably prototypal inheritance and contents traversal). Finally section 5 on page 6 discusses some other aspects of the language along with their implementation.

2. DATA TYPES AND JSON

In this section, we describe the implementation of JavaScript's dynamic type system and object notation.

2.1 Primitive Types

JavaScript has few primitive types:
`Boolean`: `true` or `false`,
`Number`: double precision floating-point values,
`String`: unicode-encoded character sequences,
`undefined`: special type with only one eponymous value.

The value `undefined` is the default value for (undeclared) variables. JSPP values are implemented as instances of a single `Value` class. This class emulates the usual boxing technique of dynamic languages, and notably stores a type flag for each value. A JavaScript variable assignment such as:

```
var foo = 1;
```

essentially translates into the following C++ code:

```
Value foo = Integer (1);
```

In JSPP, `var` is simply `typedef`'ed to `Value`, so as to preserve the original JavaScript notation. The `Value` class constructor is overloaded to handle all basics C++ types such

[1] `http://github.com/vjeux/jspp`

```
1  var undefined,
2      string = "SAC",
3      number = 4.2;
4
5  var json = {
6      "number": 42,
7      "string": "vjeux",
8      "array": [1, 2, "three"],
9
10     "nested": {
11         "first": 1,
12         "second": 2
13     }
14 };
```

Listing 1: Original JSON

```
1  var undefined,
2      string = "SAC",
3      number = 4.2;
4
5  var jsppon = {
6      _["number"] = 42,
7      _["string"] = "vjeux",
8      _["array"] = {1, 2, "three"},
9
10     _["nested"] = {
11       _["first"] = 1,
12       _["second"] = 2
13     }
14 };
```

Listing 2: JSPP JSON

as `const char*`, `int`, `double` *etc.* The argument-less version of the constructor is overloaded to create an `undefined` value.

2.2 Composite Types

In JavaScript, everything that is not of a primitive type is an object. All JavaScript objects are actually just associative arrays (or dictionaries) in which the keys are strings. Keys are often referred to as the object's "properties".

All values of a primitive type may have an object counterpart. This allows them to have properties and specific "methods". JavaScript automatically boxes primitive values when used in an object context[7]. In such a situation, the objects in question are passed by reference instead of by value[12]. In the current implementation of JSPP, primitive values are always boxed.

A JavaScript *array* is an object the keys of which are the base 10 string representation of the elements positions. The first element has the key `"0"`, the second has the key `"1"` and so on. Additionally, arrays have a special `length` property which is updated automatically when new elements (with a numerical key) are added.

As mentioned before, JavaScript introduces a purely syntactic notation for objects and arrays called JSON[4] (JavaScript Object Notation). JSON introduces two distinct syntactic forms for creating regular objects and arrays. Implementing this notation (or something close), turns out to be the biggest challenge for JSPP. The result is depicted in listings 1 and 2. Eventhough JavaScript exhibits two different syntactic forms for variadic object and array initialization, we need a single C++0x feature, called "initializer lists"[14], to implement both.

As far as the array notation is concerned, the implementation is straightforward, although one drawback is that JavaScript's square brackets [] needs to be replaced with C++'s curly braces {}. See line 8 on the corresponding listings.

The object notation, on the other hand, requires more trickery. JavaScript uses colons (`:`) to separate keys from their corresponding values. Our problem is that the colon is not an overloadable operator in C++. Consequently, we choose to use the equal sign (`=`) instead. See for instance line 6 on the corresponding listings.

Since JavaScript object properties (keys) are strings, we use the type `const char*` in JSPP. However, values of that type are not class instances, and hence cannot be operator-overloaded (something required for property initialization, as explained below). One possible notation to transform a `const char*` into an instance of a class is the following:

```
_["key"]
```

The choice of this notation is motivated by the fact that it is reminiscent of the JavaScript object property access syntax, and hence rather easy to remember:

```
obj["string"]
```

In order to implement this notation, we create a singleton class named `Underscore` equipped with an eponymous instance. This class overloads both the bracket `[]` and equal `=` operators so as to create a `KeyValue` object, that is, an object holding both a property name (the key), and its associated value.

Initializing an object contents can now be done in two ways: either by providing `KeyValue` objects, or only `Value` objects. When a `KeyValue` object is encountered, the pair is inserted directly into the object's internal map. When a `Value` object is encountered, it is regarded as an array element. As a consequence, the key is the string representation of the object's current `length` property, which is automatically increased by one after the insertion.

2.3 Caveats

From a syntactical point of view, the notation provided by JSPP is heavier than the original JavaScript one. The key part, notably, is more verbose, and equal signs are wider than colons. On the other hand, a slight improvement over JSON is that keys can in fact be any expression leading to instantiating the `Underscore` class, whereas in JavaScript, keys can only be literal strings. This allows for dynamic computation of keys, something not possible in JavaScript.

Unfortunately, using initializer lists introduces some corner-cases. For instance, constructing an object using an empty initialization list does not call the constructor with an empty initializer list but without any argument instead. This means that `{}` is equivalent to `undefined` instead of an empty object.

For technical reasons not explained here, nested objects with only one property, such as:

```
{_["nested"]= {_["prop"]=1}}
```

```
1  function forEach (array, func) {
2    for (var i = 0; i < array.length; ++i) {
3      func(i, array[i]);
4    }
5
6  };
7
8  forEach(['SAC', 2012, 'Italy'],
9    function (key, value) {
10     console.log(key, '-', value);
11
12 });
13
14 // 0 - SAC
15 // 1 - 2012
16 // 2 - Italy
```

Listing 3: JavaScript functions

```
1  var forEach = function (var array, var func) {
2    for (var i = 0; i < array["length"]; ++i) {
3      func(i, array[i]);
4    }
5    return undefined;
6  };
7
8  forEach({"SAC", 2012, "Italy"},
9    function (var key, var value) {
10     cout << key << "-" << value;
11     return undefined;
12 });
13
14 // 0 - SAC
15 // 1 - 2012
16 // 2 - Italy
```

Listing 4: JSPP functions

will produce a conflict between constructors. One solution to this problem is to explicitly cast the nested value using underscore `_` as a function: `_(val)`.

JavaScript allows both double quotes `"` and single quotes `'` for string literals (they are completely equivalent notations). JSPP only supports double quotes `"` because single quotes are used to denote characters in C++.

In JavaScript, multiple libraries can provide the same global variable (for example $), possibly overriding a previously existing one. In C++ the same situation would lead to a name clash instead. Indeed, one cannot redeclare the same variable name twice. A technique frequently used in JavaScript is to insert such a global variable as a property of the global object (`window` in the browser). When a variable is not found in any scope, JavaScript will search for it inside the global object. However, in C++ we cannot reproduce the same mechanism. A way to deal with this issue is to make an explicit local binding of the global variable, in the user code, for instance:

```
var $ = global['$'];
```

Another possibility would be to use the C preprocessor (`#define` and `#ifndef`).

In both JavaScript and C++, it is possible to declare variables in the middle of a function's body. However, JavaScript has a feature called "hoisting" which automatically removes variable declarations from a function's body and puts them back at the top of the function. As a result, the same code in JavaScript and JSPP will have different semantics. Today, hoisting has a tendency to be regarded as a misfeature[5]. Doing hoisting in C++ can be done only manually and conversely, a hoisting-less behavior in JavaScript is possible by manually adding new scopes every time a variable is declared.

Finally, JavaScript automatically creates a global variable if you assign a value to a previously undeclared variable (that is, without using the `var` keyword). This behavior is not possible in C++, although in JavaScript, it is an important source of errors and is also considered as a language misfeature. Therefore, it is not critical that JSPP does not provide it.

2.4 Optimization

JavaScript features an automatic memory management system. JSPP, on the other hand, uses a simple reference counting scheme to handle garbage collection.

Modern JavaScript implementations use boxed objects except for immediate types (such as integers). For example, V8 (the JavaScript engine that powers Google Chrome) uses tagged pointers[2] and JaegerMonkey (the one from Mozilla Firefox) does NaN boxing[18]. In our current implementation of JSPP, objects are always boxed.

Array access through a hash table is costly: keys are numbers that need to be converted into their string representation before the hash table can be accessed. This process is a lot slower than a simple offset addition. For efficiency reasons, properties with a numerical key are stored in an actual dense array in some implementations[1].

None of these optimizations are currently implemented in JSPP, although we hope to be able to provide them in the future.

3. FUNCTIONS

JavaScript functions are first-class objects, as per Christopher Stratchey's definition. The language supports passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables *etc.* Listings 3 and 4 show how functions are defined and used in both JavaScript and JSPP.

The new feature of C++0x called "lambda functions"[16] is a convenient tool to implement JavaScript function support in JSPP. One very early difficulty, however, is that the syntax is radically different. In JavaScript, a function definition looks like this:

```
function (arguments) { body }
```

On the other hand, a C++ lambda function definition is written as follows:

```
[capture] (arguments) constness -> returnType
{ body }
```

In order to implement JavaScript functions in JSPP, we define a macro called `function` which operates the transformation depicted in listing 7 on the following page:

```
1  var container = function (data) {
2    var secret = data;
3
4    return {
5      set: function (x) { secret = x; },
6      get: function () { return secret; }
7    };
8  };
9
10 var a = container ("secret-a");
11 var b = container ("secret-b");
12
13 a.set ("override-a");
14
15 console.log (a.get ()); // override-a
16 console.log (b.get ()); // secret-b
```

Listing 5: JavaScript closures

```
1  var container = function (var data) {
2    var secret = data;
3
4    return {
5      _["set"] = function (var x) { secret |= x; return undefined; },
6      _["get"] = function () { return secret; }
7    };
8  };
9
10 var a = container ("secret-a");
11 var b = container ("secret-b");
12
13 a["set"] ("override-a");
14
15 cout << a["get"] (); // override-a
16 cout << b["get"] (); // secret-b
```

Listing 6: JSPP closures

```
1  function (var n) { body }
2  // is transformed into
3  [=] (var This, var arguments, var n) mutable -> Value
4  { body }
```

Listing 7: The `function` macro transformation

We now explain how this transformation works.

3.1 Closures

Argument capture in C++ lambda functions may be done either by reference or by value. Doing it by reference corresponds to the semantics of JavaScript, but this is problematic for us because as soon as the initial variable reaches the end of its scope, it is destroyed and any subsequent attempt to dereference it leads to a segmentation fault.

Consequently, we actually have no choice but to use a by-value capture mode, denoted as `[=]`. This means that a new object is created for each lambda capturing the original variable. Since our objects are manipulated by reference, assigning a new value to an object will only update the local reference. In order to cope with this problem, we introduce a new assignment operator `|=` the purpose of which is to update all the copies (see line 5 in listing 6).

3.2 Special argument `This`

Within a function's body, JavaScript provides an implicit variable called `this`, similar to the eponymous variable in C++. Our implementation of the `function` macro (listing 7) silently inserts such a variable in the expanded lambda function's prototype in order to make it available to the programmer. Note that in order to avoid conflicts with the original `this` of C++, the internals of JSPP use `This` (with a capital T). However, an internal JSPP macro ensures that user-level code may continue to use `this` transparently.

Contrary to C++ where `this` is statically scoped however, JavaScript's `this` is being set dynamically in one of four possible ways[15]:

Function call `foo(...)`. Upon a standard function call, `this` is set to `undefined` for a strict variant of ECMAScript 5, or the global object otherwise.

Method call `obj.foo(...)`. Upon a function call resulting from an object property access, `this` is set to be that object (`obj`).

Constructor `new foo(...)`. Upon a function call used as a constructor, `this` is set to a new, empty object. See section 4.2 on the next page for more details.

Explicit `foo.call(this, ...)`. Finally, the special functions `call` and `apply` allow to explicitly set the value of `this`.

Listings 8 and 9 on the facing page demonstrate that all four ways of setting `this` are implemented in JSPP. Each JSPP variable holding a function has a hidden `"this"` property. When the function is called, the current value of the property is given to the function as its first argument (listing 7).

3.3 Special argument `arguments`

In addition to providing the special variable `this`, JavaScript also does some extra processing related to the morphology of function calls and their arguments.

Within a function's body, JavaScript provides another implicit variable called `arguments`, holding an array of all arguments actually passed to the function. Our implementation of the `function` macro (listing 7) silently inserts such a variable in the expanded lambda function's prototype in order to make it available to the programmer.

Also, when a function call does not match the intended number of arguments, JavaScript ignores the spurious ones or fills the missing ones with the `undefined` value.

Remember that functions, as any other JSPP objects, are stored in the `Value` class. All function arguments are of type `Value` as well as a function's return value. Therefore, we can distinguish the different function types based only on their arity.

In order to implement this distinction, the `Value` class is equipped with a specific typed slot, a constructor overload[17] and a parenthesis `()` operator for each arity. This imposes an implementation-dependent limit for the number of arguments to functions in JSPP.

3.4 Mutable lambda functions

By default, C++ lambda functions are assumed to be `const`. As this is in contradiction with the semantics of

```
1  function f (x, y) {
2      console.log ("this:", this);
3      this.x = x;
4      this.y = y;
5
6  };
7
8  // New creates a new, empty object
9  var a = new f (1, 2); // this: [object]
10 var b = new f (3, 4); // this: [object]
11
12 // Unbound call
13 var c = f (5, 6); // this: undefined
14
15 // Bound call
16 var obj = [42];
17 obj.f = f;
18 var d = obj.f (1, 2); // this: [42]
19
20 // Explicit call
21 var e = f.call(obj, 1, 2); // this: [42]
```

Listing 8: JavaScript's `this` variable

```
1  var f = function (var x, var y) {
2      cout << "this: " << this;
3      this['x'] = x;
4      this['y'] = y;
5      return undefined;
6  };
7
8  // New creates a new, empty object
9  var a = new (f) (1, 2); // this: [function 40d0]
10 var b = new (f) (3, 4); // this: [function 48e0]
11
12 // Unbound call
13 var c = f (5, 6); // this: undefined
14
15 // Bound call
16 var obj = { 42 };
17 obj['f'] = f;
18 var d = obj['f'] (1, 2); // this: [42]
19
20 // Explicit call
21 var e = f["call"] (obj, 1, 2); // this: [42]
```

Listing 9: JSPP's `this` variable

JavaScript, we must specify our lambda functions to be `mutable` instead (listing 7 on the preceding page).

3.5 Caveats

In regular C++ as well as in the recent lambda functions facility, every variable needs to be explicitly typed by prepending the type information before its name. As a consequence, the current implementation of JSPP requires that the programmer adds an explicit `var` type specifier in front of every function argument (compare for instance line 1 of listings 8 and 9).

In JavaScript, every function's body ends implicitly with

```
return undefined;
```

In JSPP, however, the programmer is required to add such a line manually in any function that does not otherwise return anything useful.

Finally, JavaScript has two ways of defining functions: named and anonymous. JSPP only supports the anonymous functions as lambda functions are anonymous in C++. In practice the named notation is only used for debugging purposes. We may provide some syntactic sugar to give names to JSPP functions in the future.

4. OBJECT PROPERTIES

After the tour of JavaScript functions, we now address the more general issue of object property management.

4.1 Prototypal inheritance

JavaScript features a class-less, prototype-based object model [11]. When accessing the properties of an object, JavaScript performs an upwards traversal of the prototype chain until it finds a property with the requested name. A sample JavaScript implementation of this process can be found in listing 10. We use `__proto__` to represent the link to the prototype element. This is a non-standard notation which is however supported by both SpiderMonkey (Mozilla Firefox) and V8 (Google Chrome).

```
1  function getProperty (obj, prop) {
2    if (obj.hasOwnProperty (prop))
3      return obj[prop];
4
5    else if (obj.__proto__ !== null)
6      return getProperty (obj.__proto__, prop);
7
8    else
9      return undefined;
10 }
```

Listing 10: JavaScript property access implementation

A property assignment, however, does not involve any lookup. Properties are always set directly in the corresponding object. Remember that in JSPP, property access *in both directions* (read or write) is accomplished by the bracket [] operator. In order to comply with the semantics of JavaScript, this operator needs to perform a proper property lookup, with the unfortunate consequence that the lookup in question is done for nothing in the case of a write access. Indeed, the operator itself cannot distinguish between a read of a write access.

4.2 The `new` operator

JavaScript has been designed to look like traditional object-oriented programming languages such as Java and C++. In those languages, the `new` operator creates an instance of a class. JavaScript wants to provide a similar construction, although it is a class-less language. As a consequence, the functionality of `new` in JavaScript is slightly different.

In order to emulate the functionality of a constructor which initializes the attributes of a new instance, JavaScript uses a function. This constructor function contains a property named `prototype` which links back to an object storing the methods and attributes.

The `new` operator takes that function followed by a list of arguments as arguments. It first creates an object represent-

```
1 Object.create (parent) {
2     function F () {};
3     F.prototype = parent;
4     return new F ();
5 };
```

Listing 11: JavaScript's `Object.create` function

```
1 Object["create"] = function (var parent) {
2     var F = function () { return undefined; };
3     F["prototype"] = parent;
4     return new (F) ();
5 };
```

Listing 12: JSPP's `Object.create` function

ing the instance of the class, with the `__proto__` property set to the `prototype` function. Then, it calls this function with the provided list of arguments and with the variable `this` bound to the instance being created. Finally it returns the object. A sample implementation of this process is depicted in listing 13. Since the `new` keyword is reserved in C++, JSPP uses a function named `New` (note the capital N) internally. However, an internal JSPP macro ensures that user-level code may continue to use `new` transparently.

```
1 function New (f) {
2   var obj = { __proto__: f.prototype };
3   return function () {
4     f.apply (obj, arguments);
5     return obj;
6   };
7 }
8
9 function Point (x, y) { this.x = x; this.y = y; }
10 Point.prototype = {
11   print: function () { console.log (this.x, this.y); }
12 };
13 var p = New (Point) (10, 20);
14 p.print (); // 10 20
```

Listing 13: JavaScript implementation of the `new` operator

JavaScript's `new` operator is not very intuitive and there is no standard way to access the prototype (`__proto__`) of an object. Douglas Crockford[5] has designed a small function called `Object.Create` (listing 11) which creates a new object from the specified prototype. Both `new` and `Object.create` are implemented in JSPP (the latter is shown in listing 12).

4.3 Iteration

JavaScript's `for-in` construction allows to iterate over the keys of an object. More precisely, iteration occurs over all the keys of the object itself and all the keys from its prototype chain. Properties that have the (internal) `Enumerable` property set to `false` are not enumerable. C++0x provides a new iteration facility called "range iteration"[9]. The syntax is as follows:

```
for (type element : container)
```

JSPP provides a simple macro defining `in` to expand to a colon. This lets us use range iterators directly, as shown in listings 14 and 15 on the next page.

4.4 Caveats

Our C++ implementation of property access has the same expressive power as in JavaScript, although it currently lacks some syntactic sugar. On the other hand, our syntax for iteration is completely identical to that of JavaScript.

In JavaScript, the bracket notation `obj["prop"]` is strictly equivalent to the dot notation `obj.prop`. Unfortunately, the dot notation cannot be reproduced in JSPP, which makes property access more cumbersome to both type and read.

```
1 new  Point  (1, 2); // JavaScript
2 new (Point) (1, 2); // JSPP
3
4 new  Point;     // JavaScript
5 new (Point) (); // JSPP
```

Listing 16: Syntactic overhead of JSPP's `new` operator

The `new` operator in JSPP suffers from a syntactic overhead. Because C++ does not provide any overloadable operator with a higher precedence than a function call, parenthesis are required around the constructor. It is also impossible to implement calls to the `new` operator without any argument. Those deficiencies are illustrated in listing 16.

The ECMAScript standard defines two forms for the `for-in` construction:

- **for** (**var** *variable* **in** *Expression*)
- **for** (*LHS* **in** *Expression*)

The C++0x range iteration facility does not allow for the second form however. The left-hand side expression is always a variable identifier[6].

JavaScript provides some additional operators on objects (`in`, `delete`, `typeof` and `instanceof`). The keyword `in` has two different meanings (iteration and belonging) but we cannot express both with the same keyword. As a workaround, we use the keyword `of` in the latter case (`of` has been chosen to match CoffeeScript[3]). Finally, JSPP currently provides `delete` and `typeof` as regular functions instead of operators.

5. MISCELLANEOUS

In this last section, we tackle various other aspects of JSPP in comparison with their respective JavaScript equivalent.

5.1 Code Organization

C++ does not have the notion of a "top-level". Imperative code cannot occur everywhere. It must be contained within a function's body. In order to emulate the existence of a top-level, we wrap all JSPP code inside the `main` function and a `try-catch` to handle errors. This part of the code is implemented in two header files, `javascript_start.h` and `javascript_end.h`, that the programmer is supposed to use. C++ specific includes should be added *before* `javascript_start.h`. On the other hand, JSPP specific includes should be added *after* it, as demonstrated below:

```
// C++ Includes
```

```
1  var object = {
2      "a": 1,
3      "b": 2,
4      "c": 3
5  };
6
7  for (var i in object) {
8      console.log (i, object[i]);
9  }
10
11 // a - 1
12 // b - 2
13 // c - 3
```

Listing 14: JavaScript `for-in` example

```
1  var object = {
2      _["a"] = 1,
3      _["b"] = 2,
4      _["c"] = 3
5  };
6
7  for (var i in object) {
8      cout << i << " - " << object[i];
9  }
10
11 // a - 1
12 // b - 2
13 // c - 3
```

Listing 15: JSPP `for-in` example

```
#include "../src/javascript_start.h"
// JSPP Includes
// JSPP Code
#include "../src/javascrip_end.h"
```

Compiling a JSPP program is very easy. One simply needs to compile the source file using `g++` (GNU Compiler Collection for C++) in C++0x mode:

```
g++ -std=gnu++0x jspp_file.cpp
```

At the time of this writing, `g++` version 4.6 is the only compiler to support enough of the C++0x standard to be able to compile JSPP.

5.2 Control Structures

Although C++ and JavaScript have similar control structures (`if`, `for`, `while`, `do-while`, `switch` *etc.*), some differences remain. The C++ `switch` statement only works with integers whereas JavaScript allows any data type. The `break` and `continue` instructions in JavaScript accept a label name as an argument. This allows to exit from more than one loop at a time.

5.3 Comments

JavaScript and C++ share the same comment notation. Both single-line `//` and multi-line `/* */` comments are available in JSSP.

5.4 Operators

JavaScript has many operators in common with C++. All of them can be overloaded to match the JavaScript behavior. JavaScript also has some specific operators without any C++ counterpart. The difficulty here is that it is impossible to implement them in terms of a preprocessor macro, as macros can only define new identifiers (for instance, it is impossible to define a `===` macro).

As a consequence, strict comparison operators (`===` and `!==`) are defined with two macros named `is` and `isnt` (this is in fact along the lines of what CoffeeScript[3] does). Another macro transforms `a is b` into `a * _ == b`. With the appropriate operator overloading, this behaves in compliance with JavaScript.

JavaScript provides two special operators for right unsigned bitwise shift (`>>>` and `>>>=`) and a `void` operator that gobbles its argument and always returns `undefined`. As their use is limited in practice, JSPP currently does not provide any equivalent.

5.5 Exceptions

The JavaScript exception mechanism is directly borrowed from C++, therefore we can use it directly, as shown in listings 17 and 18 on the next page, with some precautions. Most notably, we make sure to cast the thrown argument into an instance of `Value`. In order to do that, a macro defines `throw` to translate into `throw _ =`. The equal `=` operator for underscore transforms the argument into an instance of `Value`. In a similar vein, the argument to `catch` in C++ needs to be typed. A simple macro works around this in JSPP by providing the type annotation automatically.

A further extension of JSPP will be to implement a way to display the stack trace in order to improve the programmer's debugging experience.

5.6 Automatic semi-column `;` insertion

Like C++, JavaScript uses semi-columns `;` as statement delimiters. In order to make the language friendly to new developers, there is a heuristic to insert semi-columns automatically during the parsing phase in JavaScript [10]. It is impossible to implement this in JSPP without modifying the C++ compiler. Therefore, JSPP does not support automatic semi-column insertion.

5.7 Properties importation

The `with` instruction of JavaScript is a relatively unknown construct that imports all the properties of an object as local variables. Mozilla recommends against using it and ECMAScript 5 strict mode goes even as far as forbidding its use altogether. As there is no simple way to emulate this in C++, JSPP does not support it.

5.8 The `eval` function

Although `eval` is often considered as bad practice in user-level code, it plays an essential role in dynamic languages (read-eval-print loop implementation, reflexivity, dynamic code creation and execution *etc.*). Unfortunately, this is not possible to implement in C++ as the code is compiled once and for all. The lack of an `eval` function is probably the biggest downside of JSPP.

6. CONCLUSION

In this paper, we have demonstrated that in a time where the differences between static and dynamic languages are starting to fade away, the novelties from the recent C++0x standard, makes it is relatively easy to implement a JavaScript

```
var problematic = function () { throw "Exception!"; };

try {
    problematic ();
} catch (e) {
    console.log ("Error:", e);
}

// Error: Exception!
```

Listing 17: JavaScript exceptions

```
var problematic = function () { throw "Exception!"; };

try {
    problematic ();
} catch (e) {
    cout << "Error:␣" << e;
}

// Error: Exception!
```

Listing 18: JSPP exceptions

layer on top of C++, providing not only the features of the language, but also much of its original notation. The key elements of C++0x in this context are lambda functions, range-based iteration and initializer lists, a set of features directly inspired from dynamic or functional languages.

A prototype implementation of such a JavaScript layer, called JSPP, has been presented. More precisely, we have shed some light on the exact amount of the original JSON syntax we are able to reproduce, how JavaScript functions are supported, how JavaScript's prototypal inheritance scheme can be emulated and we have also outlined some other various aspects of the language support.

In terms of features, the major drawback of our current implementation is the lack of an `eval` function. In terms of syntax, we are quite close to the original JavaScript notation and it is not difficult to translate code back and forth between the two languages.

It is worth mentioning that the implementation of JSPP mostly uses regular C++ features and as such, does not heavily rely on fragile "hacks". For instance, the use of preprocessor macros is very limited (see listing 19) and only serves to fill syntactic gaps. As a consequence, both the JSPP code and potential run-time error messages are completely readable.

```
#define catch(e) catch(var e)
#define throw throw _ =
#define in :
#define function(...) [=] (var This, var arguments \
        ##__VA_ARGS__) mutable -> Value
#define is * _ ==
#define isnt * _ !=
#define of * _ <
#define this This
#define new New
```

Listing 19: JSPP macros

The current implementation of JSPP is only a prototype. Core concepts such as object iteration and operators are only partially implemented, and the standard JavaScript library is not yet available. Nothing has been done yet with respect to performance, eventhough we are already aware of many optimization techniques that we can use. Finally, further development aspects are foreseen in order to improve interoperability between JSPP and regular C++ code.

Although the project started mostly as an accidental curiosity, we think that it has some serious potential applications. For one, JSPP gives the C++ programmer a means to freely incorporate highly dynamic JavaScript-like code into a regular C++ program, hereby increasing the multi-paradigm level of the application. This is in compliance with a general trend towards offering as many paradigms as possible within the same language (especially both static and dynamic aspects), something that the Lisp family of languages has been doing for a long time, Racket being one of the most recent striking examples in this matter. Finally, just like Clojure helps bringing more people from Java to Lisp, it is possible that JSPP will help bring more people from C++ to JavaScript.

7. REFERENCES

[1] V8 issue: Wrong order in object properties iteration. `http://code.google.com/p/v8/issues/detail?id=164`.

[2] M. Ager. Google i/o 2009 - v8: High performance javascript engine. `http://www.google.com/events/io/2009/sessions/V8BuildingHighPerfJavascriptEngine.html`.

[3] J. Ashkenas. Coffeescript. `http://www.coffeescript.org/`.

[4] D. Crockford. JSON: Javascript object notation. `http://www.json.org/`.

[5] D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[6] A. Croll. Exploring javascript for-in loops. `http://javascriptweblog.wordpress.com/2011/01/04/exploring-javascript-for-in-loops/`.

[7] A. Croll. The secret life of javascript primitives. `http://javascriptweblog.wordpress.com/2010/09/27/the-secret-life-of-javascript-primitives/`.

[8] M. Felleisen. Racket. `http://racket-lang.org/`.

[9] D. Gregor and B. Dawes. N2930: C++0x range-based for loop. `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2009/n2930.html`.

[10] Inimino. Javascript semicolon insertion. `http://inimino.org/~inimino/blog/javascript_semicolons`.

[11] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPLSA '86, pages 214–223, New York, NY, USA, 1986. ACM.

[12] J. Snook. Javascript: Passing by value or by reference. `http://snook.ca/archives/javascript/javascript_pass`.

[13] B. Stroustrup. C++0x - the next iso c++ standard. `http://www2.research.att.com/~bs/C++0xFAQ.html`.

[14] B. Stroustrup and G. D. Reis. N1919: C++0x initializer list. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1919.pdf.

[15] I. Wetzel and Z. Y. Jiang. Javascript garden: How this works. http://javascriptgarden.info/#function.this.

[16] J. Willcock, J. Järvi, D. Gregor, B. Stroustrup, and A. Lumsdaine. N1968: C++0x lambda expressions and closures. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1968.pdf.

[17] A. Williams. C++0x: Overloading on lambda arity. http://stackoverflow.com/q/4170201/#4196447.

[18] A. Wingo. Value representation in javascript implementations. http://wingolog.org/archives/2011/05/18/value-representation-in-javascript-implementations.