BooJs Documentation

Release 0.0.1

Iván -DrSlump- Montes

December 19, 2015

A JavaScript backend for the Boo language.

Contents

CHAPTER 1

Overview

BooJs allows to compile Boo source code into JavaScript code for its execution in browsers and other JavaScript environments like Node.js.

1.1 Motivation

Developing large code bases with JavaScript is hard, even with the current set of tools and frameworks there are just so many times when a statically typed language would find subtle bugs at the compiling stage instead of when testing. Moreover, people from many different backgrounds are put together to develop large applications and not all of them embrace or are trained in using the good parts of the language. The situation is somewhat improved by the use of languages like CoffeeScript, although for the overhead of adding a compilation step they don't offer much more than a nicer syntax. Boo is great because it will not only give you a nicer, more structured syntax but also has a pretty intelligent compiler to help you in your development.

CHAPTER 2

Comparison with Javascript

Feature	Boo	Javascript
First-class functions	yes	yes
Function expressions	yes	yes
Closures	yes	yes
Scope	Lexical (Function, Type)	Lexical (Function)
Name resolution	Static and Dynamic	Dynamic
Type system	Static and Dynamic (Strong)	Dynamic (Weak)
Variadic functions	yes	yes (via *arguments*)
Inheritance	Class/Type based	Prototypal
Generators	yes	yes (Harmony)
List comprehensions	yes	yes (Harmony)
Iterators	yes	yes (Harmony)
Properties	yes	yes (Harmony)
Destructuring	yes	yes (Harmony)
Method overloading	yes	no
Operator overloading	yes	no
Macros	yes	no
Pattern Matching	yes	no (only limited switch)

CHAPTER 3

Comparison with Boo

BooJs strives to be as much compatible with Boo as possible, there are however a few differences, some are the result of the project design, others are mandated by performance reason and then there are a few *improvements* to the language that haven't yet be ported to standard Boo.

3.1 Type system and standard library

Probably the biggest difference is that BooJs uses the JavaScript type system and standard library instead of those in .Net. This is by design, it ensures proper performance and JavaScript's eco system of libraries is very rich anyway. Besides, the scope of the .Net library is huge and porting it to JavaScript would require an IL to JavaScript compiler while BooJs works at a much higher level.

These differences mean that you can't compile the same source files with Boo and BooJs without somehow abstracting those differences. Of course it's possible to create your own proxy objects and extension methods to emulate the JavaScript types for Boo or vice versa.

3.2 Primitive types

The following boo literal types are not defined in BooJs: char, sbyte, byte, short, ushort, long, ulong and single. Only `int`, `uint` and `double` are supported as number types.

At the runtime, standard JavaScript rules apply when working with number types, in summary all numbers are actually 64bit doubles with a 53bits mantissa, when used with binary arithmetic operators all numbers are casted to 32bit integers.

Note: There is an additional BooJs literal type with the symbol `any`. It's just an alias for `duck`.

3.3 Enums

Enums are converted to simple JavaScript literal objects mapping keys to numbers. This allows to use enums for 90% of the use cases normally, however when printing them they are just integer values, it's not possible to obtain the associated key.

```
enum Foo:
    orange
    apple = 2
```

```
print Foo.apple  # outputs "2"
```

3.4 Named parameters

Boo supports the following syntax to initialize object properties when calling a constructor.

```
f = Foo("hello", PropOne: 100, PropTwo: "value")
# Translates to:
# f = Foo("hello")
# f.PropOne = 100
# f.PropTwo = "value"
```

In BooJs it is also possible to use named parameters for plain methods, however when used with a non-constructor the parameters are converted to a Hash, there is no actual matching for the referenced method parameter names. The reason is that BooJs tries to simplify the integration with JavaScript code and there isn't a clean way to obtain the argument names for an arbitrary function.

```
foo(100, x: 10, y: 20)
# Translates to:
# foo(100, {"x": 10, "y": 20})
```

The use of a last argument accepting a hash with additional options is actually a pretty common pattern in JavaScript libraries, so this compiler transformation turns out to be very useful to produce readable code but still allowing natural integration with JavaScript code.

3.5 Generators

Check out the specific documentation about them. In essence the changes are compatible with plain Boo.

3.6 Safe Access / Existential operator

BooJs supports a new unary operator, represented by a question mark, that allows to perform two common action: check if a value is not null and protect against accessing null/undefined references.

By suffixing an expression with `?` the compiler will convert the expression to a test checking if the expression is different to null. This is good way to ensure we do proper null element comparisons, working around issues with 0 and an empty string evaluating as false in JavaScript.

If the `?` symbol appears before a dot, an open parenthesis or an open square bracket it will protect it to ensure that the next part of the expression is only accessed if the previous part is not null, otherwise a null value is returned.

Often times we have to work with nested structures, which might have some paths nulled, instead of manually checking every step of the path before accessing it we can use the safe access operator to do it for us.

```
# evaluates to a boolean by testing if `foo` is not null
foo?

# calls `foo.bar()` if foo is not null
foo?.bar()

# evaluates to null if any of the nested objects is null
```

```
if person?.address?.city == 'Barcelona':
    pass
```

CHAPTER 4

Compilation

Compilation in BooJs differs from other popular Javascript transpiler solutions in that there isn't a one-to-one relation between a .boo file and a .js file. The compiler will consume any number of .boo files to generate a single output .js file as a result of the compilation process.

The resulting javascript file is the equivalent to a .Net assembly (a .dll file) or a .so file from GCC. It contains all the generated code from the source files, no matter if it was just one or a dozen.

If no output filename is given to the compiler, it will use the name of the first source file given. See the following examples:

```
$ boojs test.boo
# Generates test.js

$ boojs test1.boo test2.boo
# Generates test1.js

$ boojs test1.boo test2.boo -o:out.js
# Generates out.js
```

If the compiler detects an error while processing the source files it will report it and exit with an exitcode of 1. A successful compilation won't output anything and terminate with an exitcode of 0. You can use this exit code values to integrate the compiler into build systems (ie: your text editor).

4.1 Response Files

Response files (.rsp) are text files that contain compiler switches and arguments. Each switch can be on a separate line, allowing comments on lines prefixed with the # symbol. To instruct the compiler to parse a response file we pass it as an argument prefixed with the `@` symbol. Response files can be nested by including an argument line with the `@` prefix and the path to another response file.

```
$ cat cool-project.rsp
# This is the configuration to compile my cool project
-embedasm-
-reference:libs/mylib.js
-o:coolproject.js
# Project files to compile
file_a.boo
file_b.boo

$ cat cool-project.verbose.rsp
-verbose+
@cool-project.rsp

$ boojs -debug @cool-project.verbose.rsp
```

Note: This mechanism is a great way to automate simple builds without using a dedicated build tool. Even when using a build tool they can simplify the integration of the BooJs compiler.

4.2 Compilation metadata

Boo is a statically typed language, in order for the compiler to check if the code complies with the type contracts it needs to know what the valid types are for using a symbol. Since Javascript is not typed, once we have compiled some Boo code the compiler wouldn't be able to apply those type checks without having access to the original .boo files. To avoid having to compile again and again all the source in your project the compiler will embed type information as a comment `//# booAssembly` in the generated javascript file, this allows the compiler to have all the needed information when referencing an already compiled library.

This type information is quite heavy in size, so it's recommended to strip it before publishing the files for its use in production, since it's just needed by the compiler and in any case used at runtime. Most javascript *bundlers* or optimizers will take care of removing comments in the javascript code, so while BooJs does not offer a tool for stripping this info it is a trivial operation if you include one of these optimizers in your build process.

4.3 Improving the code-compile-test cycle

In order to ease your development work flow you can instruct the BooJs compiler to keep watching your project files for modifications, triggering automatically a compilation on every change in the source files. By using the –*watch* command line flag, the compiler will keep running after the first compilation, monitoring for changes in all the involved files, automatically compiling a new version of the program if a change is detected.

```
$ boojs --watch -o:test.js test1.boo test2.boo
```

In this execution mode compiler messages are outputted to the `stderr` using the yellow color for warnings and the red color for errors.

To exit the watcher mode just press `ctrl-c` to terminate the compiler process.

Note: Since the compiler generates *assemblies* from the source files, it can't monitor a directory for new files and the such. If your project consists of different assemblies you will have to launch the compiler in watcher mode for each one of them, terminating and launching them again if you add, delete or rename any source file.

4.4 SourceMaps

Google Chrome and Firefox (other browsers will probably support it shortly) offer support for mapping a Javascript file with the files that were used to generate it, .boo files in our case. When we enable the sourcemap feature in the compiler two things will happen, first a new file will be generated containing a json payload with the sourcemap metadata (version 3), secondly a special comment will be included in the generated javascript file indicating the location of the sourcemap metadata.

When debugging a javascript file in a sourcemap supporting environment, we would be able to operate on the original Boo code instead of the generated Javascript one, including the option to interactively debug the program step by step based on Boo statements.

```
$ boojs -o test.js -sourcemap:test.js.map *.boo
# Generates test.js and test.js.map
```

Note: If we are using the `Boo.debug.js` runtime addon and we compile in debug mode, we will be given a processed stack trace when an uncaught exception occurs, mapping the Javascript to the original Boo code. This functionality should work independently of the native support for sourcemaps in the executing environment.

4.5 Runtime and dependencies

BooJs requires a small runtime for the generated code to work, besides any other dependencies your program may be using (ie: jQuery). These dependencies should be provided in the executing environment before loading the generated code, by default BooJs won't load them automatically or include them as part of the generated file.

At the bare minimum you will need to make sure that the `Boo.js` file has been loaded. There is an optional runtime `Boo.debug.js` file which can help in debugging problems, which you can use while developing.

4.6 Generating code for production

The compiler generates performant code by default if you don't use the `--debug` switch. However it tries to generate Javascript code that is easy for a human to read, in order to ease troubleshooting problems. If you're targeting an environment where size matters (web site, mobile devices, ...) you will most probably benefit from using a Javascript optimization tool like Google Closure or UglifyJS.

These tools will first of all remove metadata included in the form of comments which is only needed by the compiler. Moreover they will mangle variable names to make them shorter and thus reducing the final code size. Some of them will even reduce the size by removing dead code (not used types for example).

Note: The compiler will try to generate code that is safe to process thru any of these optimizers, so you won't have to worry about configuring them to produce a valid result.

CHAPTER 5

Macros and Syntactic Attributes

By default some macros are made automatically available for their use in Boo without importing any namespace. Some of them are equivalent to standard Boo while others are only available when using the BooJs compiler.

Note: Macros, Meta-Methods and AST Attributes are resolved at compile-time, this means that they are executed as part of the compilation and thus not part of the generated JavaScript code. They work by transforming the program syntax tree, refer to the standard Boo documentation to learn more about them.

5.1 Macros

5.1.1 assert

Use this macro to ensure some condition applies when compiling in debug mode. If the given condition fails it will raise a `Boo.AssertionError` exception. When compiling without the debug switch the assertion is removed from the generated source code.

```
assert arg > 10
# Raises Boo.AssertionError('arg > 10')
assert arg < 100, 'Argument must be less than 100'
# Raises Boo.AssertionError('Argument must be less than 100')
```

5.1.2 const

Boo's syntax doesn't allow to define variables at the module level, the compiler will interpret such declarations as the start of the module entry point. This macro allows to work around this issue and bind static variables to the current module.

```
namespace MyNS

const foo = 10
# Declares MyNS.foo as an int with a value of 10
const foo as string = 'foo'
# Declares MyNS.foo as a string with a value of 'foo'
```

5.1.3 global

Unlike JavaScript, Boo's compiler will complain if you reference a symbol that hasn't been previously declared either in the current module or imported from another namespace. In order to integrate Boo code with external symbols defined somewhere else in your execution environment, the `global` macro provides the means to make those symbols available in the code.

```
global jQuery   # jQuery is available with a type of duck
jQuery('#foo').html('Hi there!')

global MY_FLAG as int   # MY_FLAG is available with a type of int
print MY_FLAG + 10
```

5.1.4 ifdef

Allows to define blocks for conditional compilation by evaluating the condition against the compiler defined symbols. You can use your own defined symbols with the ‘‘-D:symbol‘.

If the condition evaluates to false the contents of the block are removed from the compilation unit.

```
ifdef DEBUG:
    print "Debug mode enabled"

ifdef not WINDOWS and DEBUG:
    print "Compiling on a non-windows system"
```

5.2 js

Every now an then there is the odd case where we can't map some JavaScript code to BooJs, or perhaps we are just prototyping something and we want to copy-paste some snippet of code. The `js` meta method will include any literal string given as argument without modifying it. Any other expression will be wrapped in a call to `eval`.

```
a = 100 + js('10')
js `alert(a)`
# generates:
# var a = 100 + 10
# alert(a)

# We can include multi line snippets too
a = js(`
    [ 'foo',
      'bar'
    ]
`)

# Anything other than a string literal is generated with a call to eval
a = 'alert("foo")'
js a
# generates:
# var a = 'alert("foo")';
# eval(a);
```

5.2.1 match

BooJs automatically exposes the `match` macro from Boo.Lang.PatternMatching. This macro allows to use pattern matching in your code completely at compile time. You can learn a few of the basics from this mailing list message.

5.2.2 new

JavaScript allows to instantiate new objects in a variety of ways, when interfacing external code without using type definitions for it we may need to indicate the compiler how it should call a constructor function.

The `new` symbol is not actually a macro but a *meta-method*, the syntax for applying it is the same as for functions but it's resolved at compile-time, so it doesn't appears in the generate JavaScript code.

Since Boo already has a `new` keyword, used to define members with the same name as one in the inherited type, we can't use it directly to flag a constructor. To use it we have to prefix `new` with `@` to tell the compiler that we are referencing the meta-method instead of the keyword.

```
global Coords

obj = Coords(10, 20)
# js: obj = Coords(10, 20);

obj = @new( Coords(10, 20) )
# js: obj = new Coords(10, 20);
```

5.2.3 preserving

Solves the common problem of temporally backing up some variables to perform an action.

```
x = 'foo'
y = [10, 20]
preserving x, y[1]:
    x = 'bar'
    y[0] = 50
    y[1] = 60

print x     # 'foo'
print y     # [50, 20]
```

5.2.4 print

The `print` macro outputs the given arguments using the `console.log` function if available in your environment.

```
foo = 'DrSlump'
print "Hello", foo    # Hello Drslump
```

5.2.5 trace

Very similar to `print` but only outputs when in *debug* mode. The message is prefixed with the filename and line number where the macro was used.

```
trace 'hello there'    # filename.boo:11 hello there
```

5.2.6 with

Even though the `with` statement is considered evil in modern JavaScript, this macro serves a similar purpose avoiding the drawbacks of its JavaScript sibling. It sets a value as default target for expressions without one but does so explicitly by prefixing the expressions with a dot.

```
with jQuery:
  .each({x| print x})    # Converted to jQuery.each()
  each()                 # Looks for a method named "each"

with foo = jQuery('#foo'):
  .html('Hi there!')     # Converted to foo.html('Hi there!')
```

5.3 Attributes

5.3.1 Extension

Like in C# it's possible to *extend* a type with new methods without modifying the type's hierarchy chain. The first argument of the method defined as a extension is the type to which that method should be attached. If the compiler doesn't find a proper method defined in the extended type it will check the extensions for a proper match.

```
[extension]
def toISO(date as Date):
  return date.getFullYear() + date.getMonth() + date.getDate()

[extension]
def incr(date as Date, seconds as int):
  date.setTime( date.getTime() + seconds*1000 )

d = Date()
print d.toISO()      # Converted to: print toISO(d)
d.incr(3600)         # Converted to: incr(d, 3600)
```

CHAPTER 6

Overloading

6.1 Method overloading

> Allows defining several functions with the same name which differ from each other in the types of the input arguments.

Boo method overloading mechanism takes into account differences in the input arguments of methods with the same name, it does not take into account differences in the return types of those methods. Moreover, when looking for the best candidate in an instance, it will choose one from the target instance type and will only look for candidates in inherited types if it couldn't find one.

Each overloaded method is assigned an unique suffix, so an overloaded method named `foo` with two different signatures will generate two additional methods named `foo$0` and `foo$1`. The compiler will try to find the best candidate at compile time, however there are times when that's not possible, for instance when the target object is *ducky*, performing the resolution at runtime. This has obviously a cost, so try to avoid mixing overloading and duck typing in performance critical sections.

When the overloaded method is public and is used from external code, the calling site cannot be updated to target a specific version of the method. Instead, calls to the method make use of a runtime mechanism to forward the call to a valid candidate.

Note: The current implementation of the runtime dispatching only takes into account the number of arguments, not their types. This shortcoming will be removed in the future, implementing a more advanced resolution.

CHAPTER 7

Runtime library

BooJs requires a small runtime library to support the execution of the compiled code. This library is located in the `Boo.js` file with an approximate size of 4Kb when minified.

7.1 Reasoning for using a runtime

Among the Javascript community there is the idea that requiring a companion runtime library in order to run the generated code is a bad decision. The major complain is that it imposes an additional dependency to keep track of, also increasing the final code size of the delivered code, when for a *Hello World* style program you must include a few extra kilobytes of functions that will probably never be used.

Obviously it would be ideal to avoid the use of a runtime library, unfortunately it's not possible to do. In order to keep code compatible with the original semantics from Boo, some helper functions must be available, specially since Boo does not have a 100% static type system.

Given the fact that a runtime is actually needed, it could be reduced to include just the minimum functions necessary, however there is some stuff that is much easier to implement with a small runtime function than it would be to statically generate code for from the compiler. So the approach BooJs follows is to try to keep the runtime library small but not at the expense of complicating the compiler excessively.

The complaint about having an additional dependency is easily resolved by including the runtime code inside the compiled file. It's very easy to do for Javascript and can be automated with any of the several build tools available. For the one regarding the increased code size, let's just say that if the compiler had to generate code to support the language semantics, the final size of all that feature specific code will probably exceed the size of the runtime for medium sized projects. It makes no sense to measure the overhead of a runtime using a simple example application, for developments so simple and small it probably doesn't make sense to use anything other than vanilla javascript for them.

Many compilers to Javascript will actually output their runtime support functions as part of the generated code, so even if they can be advertised as runtime-free, it's actually being included automatically by the compiler when generating code needing it. That's one approach that BooJs will probably use in the future, acting like a *linker* to just include those helpers actually needed for a compilation unit, although even then it will still have a separated runtime library with stuff that is very commonly used.

7.2 Builtins

Most of the standard Boo builtins are supported in BooJs. When referencing them from Boo code they are global symbols, if you need to reference them from JavaScript code they are available in the `Boo` object (eg: `Boo.range`).

7.2.1 __RUNTIME_VERSION__

Stores the version of the runtime currently in use. You can use this value to work around versioning issues in your code to make it compatible with different BooJs releases.

Note: To obtain the version of the compiler used to generate the code you can use the `__COMPILER_VERSION__` reference. The compiler will automatically convert those references to a string containing the compiler version.

7.2.2 Array

While arrays are directly mapped to JavaScript the compiler will offer some additional functionality when working with them: equality, addition, multiplication and membership test.

```
l = ['foo', 'bar', 'baz']
if l == ('foo', 'bar', 'baz'):
    print "all items are equal"

r = l + (10, 20)
# result: ['foo', 'bar', 'baz', 10, 20]

r = l * 2
# result: ['foo', 'bar', 'baz', 'foo', 'bar', 'baz']

if 'bar' in l:
    print "array contains 'bar'"
```

7.2.3 array

Create an array of a given type, indicating how large it is or initialize it with the values from an iterable.

```
# an array with room for 5 strings
r = array(string, 5)
# result: [ '', '', '', '', '' ]


# an array from an iterable
l = x * 10 for x in range(3)
r = array(l)
# result: [ 0, 10, 20 ]

# an array casting iterable values to a string
l = x * 10 for x in range(3)
r = array(string, l)
# result: [ '0', '10', '20' ]
```

7.2.4 AssertionError

Specific error type used by the `assert` macro.

7.2.5 CastError

Specific error type used to signal failures when casting values.

7.2.6 cat

Concatenates a list of iterables.

```
c = cat([10,20], ['a', 'b'])
# result: [10, 20, 'a', 'b']
```

7.2.7 enumerate

Obtain an array of key-value pairs from an enumerable. This is usually used to access the index value when using a `for` loop.

```
l = ('foo', 'bar', 'baz')
for idx, val in enumerate(l):
    print "$idx: $val"
# outputs: 0: foo, 1: bar, 2: baz
```

7.2.8 filter

Apply a function to an iterable to filter out items from it in the generated array. The callback function is called for each element of the iterable, if it returns a truish value them it's included in the result, otherwise the element is ignored.

```
l = range(5)
r = filter(l, { _ % 2 })
# result: [0, 2, 4]
```

7.2.9 Hash

Type to model a *hash map*, while a JavaScript's object type does work like a hash map by default, having a light weight type to reference in our code allows to easily tell apart those values for which we don't have a specific type from those that are actually expected to work with hash map semantics.

Note: Since we strive for a light weight implementation by using JavaScript object semantics, the Hash doesn't accept arbitrary types as keys. Basically keys should be restricted to string types, as they are in plain JavaScript code.

The generated code is optimized to avoid using the Hash type methods when possible, generating instructions operating with plain JavaScript object syntax. There are however the following helper methods that do not have a direct translation:

```
# Create a new Hash and initialize it with some values
hash = Hash(foo: 'Foo', bar: 'Bar', baz: 100)
# js: {foo: 'Foo', bar: 'Bar', baz: 100}

# Create a new Hash and initialize it with some key-value pairs
hash = Hash(('foo' + i, i) for i in range(3))
# js: {foo0: 0, foo1: 1, foo2: 2}

# Iterate over the list of keys in the Hash
for k in hash.keys():
    print k

# Iterate over the list of values in the hash
for v in hash.values():
    print v

# Iterate over the list of key-value pairs in the hash
for k, v in hash.items():
    print "$k = $v"

# Check if a key exists in a hash (uses JavaScript `.hasOwnProperty`)
if 'foo' in hash:
    print 'foo exists'
```

7.2.10 join

Joins the elements of an iterable to form a string applying an optional separator. If no separator is given it defaults to a single white space character.

```
l = ('foo', 'bar', 'baz')
print join(l)
# outputs: "foo bar baz"

print join(l, ', ')
# outputs: "foo, bar, baz"

print join(l, '')
# outputs: "foobarbaz"
```

7.2.11 len

Obtains the length of a string, array or Hash value. It will obtain the length of anything that exposes a length property or method. Alternatively, for objects it will report the number of own properties on them.

```
l = len([1, 2, 3])
# result: 3

l = len({'foo': 'Foo', 'bar': 'Bar'})
# result: 2

l = len('hello')
# result: 5
```

7.2.12 map

Apply a function to every element in an iterable and returns an array with the results.

```
l = ('foo', 'bar', 'baz')
r = map(l, { _.toUpper() })
# result: [ 'FOO', 'BAR', 'BAZ' ]
```

7.2.13 NotImplementedError

Specific error type raised when an abstract method is not implemented

7.2.14 range

The primary loop construct in Boo is the `for` statement, unlike the versions found in C derived languages it's not possible to indicate initialization and loop conditions, it always work by obtaining elements from an iterable. The `range` builtin generates iterables that implement most common loop cases with ease.

When a single argument is given it generates an iterable from 0 upto, but not including, the argument given.

Two arguments indicate an start number (included) and an end number (not included).

Three arguments work as with only two but the third one indicates how the stepping is done. By default it steps by 1 but we can use any value here, using a negative one for example allows to generate a decreasing iterable.

Note: The BooJs compiler will optimize `range` based loops if it's defined as the iterable in the `for` construct (eg: not assigned to a temporary variable), so its performance matches JavaScript's native `for` construct.

```
for i in range(5):
    print i
# outputs: 0, 1, 2, 3, 4

for i in range(2, 5):
    print i
# outputs: 2, 3, 4

for i in range(2, 10, 2):
    print i
# outputs: 2, 4, 6, 8

for i in range(10, 5):
    print i
# outputs: 10, 9, 8, 7, 6

for i in range(10, 5, -2):
    print i
# outputs: 10, 8, 6
```

7.2.15 reduce

Apply a function to every element in an iterable to return a final value. The callback function receives two arguments, the accumulated value and the next item from the iterable, the value returned is used as the accumulated value for the next call.

If not initial value is given it defaults to the first element of the iterable, making the first call to the function using it as accumulator and the second element of the iterable.

```
l = range(5)
r = reduce(l, { x, y | x + y })
# result: 10 (0 + 1 + 2 + 3 + 4)

r = reduce(l, 10, { x, y | x + y })
# result: 20 (10 + 0 + 1 + 2 + 3 + 4)
```

7.2.16 reversed

Obtains an array from an iterable where the elements are in inverse order.

```
l = range(5)
r = reverse(l)
# result: [4, 3, 2, 1]
```

7.2.17 String

The string type is directly mapped to JavaScript, there are however a couple of additions included by the compiler: Multiplication and Formatting.

```
s = "Foo"
r = s * 3
# result: "FooFooFoo"

r = "Foo {0}" % ('Bar',)
# result: "Foo Bar"

r = "Foo {0} {{escaped}} {1}" % range(2)
# result: "Foo 0 {escaped} 1"
```

7.2.18 zip

Builds an array of arrays by fetching an element for each of the iterables given as arguments. The algorithm stops when any of the iterables is exhausted, making it safe for using it with infinite generators.

```
names = ['John', 'Ivan', 'Rodrigo']
webs = ['foo.com', 'bar.com', 'baz.com']
r = zip(names, webs)
# result: [ ['John', 'foo.com'], ['Ivan', 'bar.com'], ['Rodrigo', 'baz.com'] ]

# This creates a Hash
h = Hash(zip(names, webs))
# result: { 'John': 'foo.com', 'Ivan': 'bar.com', 'Rodrigo': 'baz.com' }

# Get 3 random numbers (`random_generator` is a never ending generator)
for i, random in zip(range(3), random_generator()):
    print random
# outputs: 3 random numbers
```

7.3 Events

Boo Event's are a way to easily setup delegates in classes, implementing the observer pattern. Basically they allow registering a callback on them from outside the class but only firing them from inside the class.

Since it's not clear how to map this to JavaScript there is a very lightweight runtime support for them. Every event field is mapped to a function that triggers it when called, exposing two additional methods `add` and `remove` to handle subscriptions. This is transparent when using Boo code, adding a subscription is done with the += operator and removing one with −=.

```
class Foo:
    event click as callable()
    def DoClick():
        click()
```

```
f = Foo()
f.click += def ():
    print "Clicked!"
```

To use it from JavaScript code we can use the runtime interface directly:

```
f.click.add(function () { console.log('Clicked!') })
```

CHAPTER 8

Modules

Boo follows a strict mapping of one module per file, in other words, each source file is a module. From its .Net roots however there is also the concept of namespaces, which allow to expose multiple modules under a single export point.

In BooJs that mechanism is respected and mapped to what is known as the AMD pattern in the Javascript world. The reason why this pattern was chosen instead of something like Node.js `require` is that BooJs code can be targeted to run on a browser too, where synchronous loading of code is not widely supported.

Here is an example Boo module and the generated Javascript code with annotations:

```
namespace example

import myapp

def foo(s):
    notify(s)

foo("Hello")
```

```
// Namespace serves as ID and is mapped to exports so we can augment it
// Boo runtime is always passed as a dependency
// Imports are passed as additional dependencies
Boo.define('example', ['exports', 'Boo', 'myapp'], function (exports, Boo, myapp) {
    // Type definitions of the module
    function foo(s) {
        myapp.notify(s);
    }
    // Public types are exported
    exports.foo = foo;
});

// Namespace is mapped to exports
// Boo runtime is again always passed as a dependency
// Imports are passed as additional dependencies
Boo.require(['example', 'Boo', 'myapp'], function (exports, Boo, myapp) {
    // Executable portion of the module
    exports.foo("Hello");
});
```

One difference with the AMD spec is that `define` and `require` are not global symbols but instead are referenced from the Boo runtime (eg. `Boo.define`). This is done to avoid a dependency or conflict with an AMD loader in the environment, BooJs includes all the needed functionality to manage AMD style dependencies. If you wish to use a more powerful loader you can just point `Boo.define` and `Boo.require` to require.js for example.

BooJs default AMD loader does not automatically fetch dependencies from disk or a web server, it expects all the dependencies to be loaded up front, its job is only to resolve them in the correct order. Since in BooJs the deployable unit is an *assembly* and not a *module* this works quite well, you just need to remember to load all the generated assemblies in your environment. For automatic loading of dependencies you can easily integrate with require.js or any other module loader that conforms to the AMD pattern.

Note: For Node.js environments a custom wrapper for the loader is in the roadmap, it will take care of automatically importing referenced dependencies.

To call BooJs modules from your Javascript code you can use `Boo.require` to obtain the desired module.

```
var example = Boo.require('example');
example.foo("Hello");
// ... or ...
var foo = Boo.require('example').foo;
foo("Hello");
```

CHAPTER 9

Generators

Generators are really powerful in BooJs, they differ from standard Boo (or C#) and model instead the pattern found in Python or Mozilla's JavaScript.

In summary, every generator becomes a *coroutine* not only able to halt execution at arbitrary points in order to return a value but also to receive values and exceptions from the outer context. These features make generators a great primitive for co-operative multitasking and event driven programming in general, for instance the Async library is built on top of generators.

JavaScript engines, with the exception of Mozilla's and recent V8 builds, do not offer native support for this kind of generator. The BooJs compiler will instrument the code, converting the generator to a state machine able to handle halting and resuming execution at arbitrary points. While the generated code is convoluted it shows to be pretty fast on modern browsers, it runs roughly at half the speed of an user land `forEach` implementation and about 70% the speed of Mozilla's native generators.

9.1 Generator interface

BooJs exposes Mozilla's iterator and generator interfaces since they are being standardized in ES6 (aka JavaScript Harmony) there is a chance that in the future they get adapted to closely follow the standard. Basically a generator returns a `GeneratorIterator` which implements `Iterator` for `next()` and also offers `send(value)`, `throw(error)` and `close()`.

9.2 Native support

The compiler only targets standard JavaScript 1.5, not generating alternative code paths for specific browsers. This is something that will probably change in the future but currently generators always get instrumented and thus are not as performant as native implementations, although they aren't as slow as they might seem!

Even if native support is not used, BooJs generators offer a compatible API and hence should work properly on every environment, including their use with native generator loop constructs.

9.3 Closing generators

Generators keep state and allow to use `ensure` (aka finally) blocks inside them so there is a need to properly close and dispose them. Boo `for` loop construct understands the GeneratorIterator interface and is able to close them when the iteration is over. However when manually iterating them you are responsible for properly closing the generators.

```
# Infinite generator
def gen():
    i = 1
    try:
        while true:
            yield i++
    ensure:
        print 'exited'

# Automatically closed by the compiler
for _, i in zip(range(3), gen()):
    print i
# Outputs: 1, 2, 3, 'exited'

# Manually closing the generator
g = gen()
for i in range(3):
    print g.next()
# Outputs: 1, 2, 3
g.close()
# Outputs: exited
```

CHAPTER 10

Async library

BooJs includes a simple yet powerful asynchronous library modeled after the Promises/A CommonJS spec also known as *thenables*. You can read more about this asynchronous programming pattern on Wikipedia.

By supporting the *Promises/A* spec we ensure compatibility with some of the most popular JavaScript frameworks like Dojo or jQuery. Other frameworks implementing the *deferred*, *promise*, *future* or *task* patterns can be easily modified to be made compatible with *Promises/A* for most use cases.

The Async library is exposed as an optional namespace, not included by default by the compiler, which you can use in your own code just by importing the `Async` namespace and loading the `Boo.Async.js` file in your environment.

10.1 Deferred

The `Deferred` class allows to create, control and resolve *promises* which can be consumed by your own code or passed on to third party libraries that understand the *Promises/A* spec.

```
def make_async(v):
    # Create a new deferred
    defer = Deferred()
    # Launch the async job
    setTimeout({
        # Resolve the deferred with the final value
        defer.resolve(v)
    }, 1000)
    # Return the deferred promise which can be observed
    return defer.promise

# Obtain a promise
promise = make_async('foo')
# Register a callback to observe the successful resolution of the promise
promise.done({ x | print x })
# Register a callback to observe the wrongful resolution of the promise
promise.fail({ x | print 'Error:', x })
```

Note: Unlike some implementation that focus on raw performance (ie jQuery), `Deferred` works internally as a tree instead of a list. Each time you attach a callback to a deferred a new one is created internally returning its public interface, a `Promise`, this allows to model complex flows avoiding side effects.

Note: A common pitfall when using the promise pattern is that some errors might go unnoticed if we are not very careful to observe failures on every promise generated. To avoid this you can assign a global callback in

`Deferred.onError` to act upon any rejected promise that isn't explicitly controlled. It's the equivalent to a handler for uncaught exceptions. By default, if no custom handler is assigned, a exception is raised with the reported error.

10.2 Promise

A `Promise` object is the public interface of a `Deferred`, there is a 1:1 relationship between them. While a `Deferred` allows to control its cancellation, rejection and resolution a `Promise` only allows to register our interest in its resolution (successful or not).

The *Promises/A* spec just specifies that a `Promise` object must expose a public method called `then` which receives 3 arguments: *successCallback*, *failureCallback* and *progressCallback*. This simple design makes it trivial to share promises between third party libraries, like jQuery for instance, allowing to observe the resolution of an asynchronous task with ease.

10.3 Utilities

10.3.1 enqueue

This simple method allows to defer the execution of a callback until the stack is empty. This is specially useful when you want to trigger an action just after a configuration step has completed. For instance, this method is used internally by `Deferred` to resolve them, that's why they can be used also with immediate values.

```
d = {}
enqueue({ d.run() })
d.run = def():
  print 'Foo!'
# d.run() will called now
```

Note: In a browser environment it will use `setImmediate` or `setTimeout` with a timeout of 0. For Node it will use `process.nextTick`.

10.3.2 when

A common pattern is to wait until two or more action have completed before continuing. The `when` method will produce a `Promise` that gets resolved only when all the arguments given are resolved successfully. If any of them is rejected the `Promise` is rejected also and the other arguments are *canceled*.

```
p = when( jQuery.get('/data/a'), jQuery.get('/data/b'), 'immediate value' )

p.done def(results):
  a, b, immediate = results
  print a, b, immediate

p.fail def(error):
  print 'An error occurred fetching data:', error
```

Note: `when` can also be used as a shortcut to wrap any value in a promise which gets almost instantly resolved.

10.3.3 sleep

This method generates a `Promise` that gets resolved after the given milliseconds. It can be used to delay the execution of some code without blocking the execution thread.

```
p = sleep(10s)
p.done:
    print 'Woke up after 10 seconds'

# It also supports providing a callback directly
sleep 10s:
    print 'Woke up after 10 seconds'
```

10.4 Async/Await

One of the nicest features of the Async library is its implementation of the Async/Await pattern. Modeling your logic around *promises* is a nice way to support asynchronicity, however it forces you to replace the language native flow control mechanisms by those of the `Deferred` API. The Async/Await pattern removes that limitation, allowing you to write *promise* based code as if they were synchronous operations.

Under the hood the pattern makes use of coroutines (constructed via generators) to suspend and resume the execution of code at any point in a function based on the result of a `Promise`.

When we annotate a method as `async` we are telling the compiler that we want to control its execution in a special way, suspending it when an `await` keyword is found until its value is resolved. In other words, the `await` keyword indicates that we want to wait at that point until the given `Promise` object is resolved, avoiding the need to chain callbacks to control the program logic flow.

```
[async] def fetch(url):
    print "Fetching $url"
    try:
        # jQuery's ajax methods are Promises/A compatible
        await data = jQuery.get(url)
        print data
    except ex:
        print 'Error:', ex
```

The code above is roughly equivalent to the following one:

```
def fetch(url):
    print "Fetching $url"
    promise = jQuery.get(url)
    promise.done = def(data):
        print data
    promise.fail = def(error):
        print 'Error:', error
```

Even in this simple example the benefits of the Async/Await version are obvious. The complexity of using the promise API is hidden from us, with the added benefit that every *async* method always returns a *promise* itself, thus it's very easy to compose complex flows with them.

```
def fetch(id):
    print 'Fetching data'
    await data = jQuery.get('http://ajax.com/' + id)
    return data

def update(id):
    await data = fetch(id)
    data.foo = 10
    await jQuery.put('http://ajax.com/' + id, data)
    print 'Data updated'
```

Another point where this pattern excels is in the handling of error conditions. There is no need to observe the promises for failures, using the native try/except mechanism we can control failures in a clean way, even maintaining a meaningful stacktrace to troubleshot any problem.

Note: The `await` keyword also works for multiple values, by using `when` under the hood. This means that we can easily parallelize asynchronous operations and only resume execution when all of them have completed.

Resources

CHAPTER 11

Indices and tables

- genindex
- modindex
- search